1

IDA DOCUMENT D-1425

PROCEEDINGS OF THE WORKSHOP ON LARGE, DISTRIBUTED, PARALLEL ARCHITECTURE, REAL-TIME SYSTEMS

Dennis W. Fife, *Task Leader*

Norman R. Howes
Jonathan D. Wood

July 1993

Sponsored by
Ballistic Missile Defense Office
and
NASA Ames Research Center

IDA

INSTITUTE FOR DEFENSE ANALYSES
1801 N. Beauregard Street, Alexandria, Virginia 22311-1772

DEFINITIONS

IDA publishes the following documents to report the results of its work.

Reports

Reports are the most authoritative and most carefully considered products IDA publishes. They normally embody results of major projects which (a) have a direct bearing on decisions affecting major programs, (b) address issues of significant concern to the Executive Branch, the Congress and/or the public, or (c) address issues that have significant economic implications. IDA Reports are reviewed by outside panels of experts to ensure their high quality and relevance to the problems studied, and they are released by the President of IDA.

Group Reports

Group Reports record the findings and results of IDA established working groups and panels composed of senior individuals addressing major issues which otherwise would be the subject of an IDA Report. IDA Group Reports are reviewed by the senior individuals responsible for the project and others as selected by IDA to ensure their high quality and relevance to the problems studied, and are released by the President of IDA.

Papers

Papers, also authoritative and carefully considered products of IDA, address studies that are narrower in scope than those covered in Reports. IDA Papers are reviewed to ensure that they meet the high standards expected of refereed papers in professional journals or formal Agency reports.

Documents

IDA Documents are used for the convenience of the sponsors or the analysts (a) to record substantive work done in quick reaction studies, (b) to record the proceedings of conferences and meetings, (c) to make available preliminary and tentative results of analyses, (d) to record data developed in the course of an investigation, or (e) to forward information that is essentially unanalyzed and unevaluated. The review of IDA Documents is suited to their content and intended use.

The work reported in this document was conducted under contract MDA 903 89 C 0003 for the Department of Defense. The publication of this IDA document does not indicate endorsement by the Department of Defense, nor should the contents be construed as reflecting the official position of that Agency.

IDA DOCUMENT D-1425

PROCEEDINGS OF THE WORKSHOP ON LARGE, DISTRIBUTED, PARALLEL ARCHITECTURE, REAL-TIME SYSTEMS

Dennis W. Fife, *Task Leader*

Norman R. Howes
Jonathan D. Wood

July 1993

INSTITUTE FOR DEFENSE ANALYSES

Contract MDA 903 89 C 0003
Task T-R2-597.2

Preface

This document was prepared by the Institute for Defense Analyses (IDA) under the Task Order, Ballistic Missile Defense Office Software Technology Plan, and fulfills an objective of the task, to deliver the proceedings of the workshop on Large, Distributed, Parallel Architecture, Real-Time Systems. The workshop was sponsored by the Ballistic Missile Defense Office and NASA Ames Research Center.

The Document was assembled and the introductory remarks were written by Dr. Dennis Fife, Dr. Norman Howes, Mr. David Wheeler, and Mr. Jonathan Wood. The contents of the document were furnished by the participants of the workshop, to whom we express our sincere appreciation.

Foreword

The Institute for Defense Analyses hosted a workshop on the software engineering issues associated with large, distributed, parallel architecture, real-time systems from March 15th through March 19th, 1993. It was jointly sponsored by the Strategic Defense Initiative (now the Ballistic Missile Defense Organization) and NASA Ames Research Center. The workshop's purpose was to bring together researchers from industry and academia who are currently designing systems of this type or doing research that bears upon real-time issues of interest to the sponsors. Participation at the workshop was by invitation. The number invited was small (twenty-three). The number accepting was seventeen. They contributed a total of fifteen papers. Also participating were representatives from the sponsoring agencies.

The participants were asked to submit a position paper prior to the end of 1992 on the following four issues: (1) What is the best design methodology for this class of systems? (2) What is the proper relationship between design theory and scheduling theory? (3) What is the best method for validating this class of systems? and (4) What are the most promising areas where resources might be applied for near term benefits? These position papers were distributed to all the participants on February 1, 1993 in order for the participants to review the positions being taken by their fellow participants. At the workshop, each participant was given the opportunity to formally present his or her position. However, much of the workshop time was devoted to informal discussions of the basic issues.

The proceedings that follow include several items. First, there is a copy of the workshop agenda (which was not followed in exact order due to several on-line reschedulings of the formal talks). Second, there is a summary of the discussions of the four major issues of the workshop. This summary was prepared by IDA workshop participants and thus may not reflect the opinions of all the participants. Third, there is a copy of the fifteen position papers presented at the workshop. Fourth, there are copies of the transparencies for most of the formal presentations. Copies of presentations by the sponsors were not included, at their request.

The workshop was very helpful in summarizing and clarifying the key issues related to the class of systems discussed at the workshop. It was also helpful in quantifying the magnitude of the problems associated with the issues. On behalf of the sponsors and IDA, we would like to express our thanks to all participants for sharing with us their experience and insights.

Dennis Fife

Norman Howes

David Wheeler

Jonathan Wood

Summary of the Workshop

The workshop revealed sharp differences of opinion with regard to the first two workshop issues, namely, what is the best method to design large, distributed real-time systems, and what should be the relationship between design theory and scheduling theory? The differences confirmed the often mentioned "disconnect" between design theory and scheduling theory. Several real-time designers at the workshop indicated that they concentrate on designing practical systems that have acceptable real-time behavior, relying on design guidelines they have developed from experience that are meaningful to them. These guidelines often have provisions for designing functionality into the system other than timing behavior, e.g., maintainability, conceptual clarity, testability, etc. They then use measurement and tuning during the software testing phase to correct or improve their design's real-time behavior.

Scheduling theorists at the workshop thought that designers should concentrate on deriving abstract representations of processing in order to apply an established scheduling model, e.g., rate monotonic scheduling, that can ensure acceptable real-time behavior (if the abstract representation faithfully models the real system). If the real system fails to satisfy the necessary assumptions and approximations of the model, then the scheduling theorists advised redesign of the application system to better fit the abstract model. This disconnect needs to be overcome through better design methods and tools that improve understanding and control of real-time behavior much earlier than software testing, and through scheduling concepts that support more general and flexible decision-making. The workshop discussion tried to identify specific strategies to "work-around" the disconnect and thereby make best use of the guidance available from both schools of thought. Basically, these work-around strategies were iterative techniques to do either a schedulability analysis of a design or a designability analysis of an abstract scheduling representation (depending on which point of view one adopts) early in the design phase, and to continue refining the design with schedulability considerations as the design progresses.

On the other hand, the workshop revealed consensus on how real-time systems should be validated. This consensus called for designing code instrumentation into systems for validation purposes and leaving it in when a system is fielded. This insures that the system timing is not altered between validation and fielding and it provides a "window" into the operational system to allow reproduction of event sequences leading up to any system failures. This is important since real-time system failures are usually difficult to reproduce due to concurrent activities within the system.

The workshop provided some ideas for investigations in the near term that address issues arising with the new generation of distributed (and often parallel) real-time systems. Specifically, it was recommended that the proper application of object oriented technology be determined for real-time systems and that a standardized real-time Inter-Process Communication (IPC) mechanism be developed that is independent of system architecture. Further, it was recommended that there should be more support (funding) for experimentation with (as opposed to creating more) real-time design methodologies, in order to quantify their value by competitive design "shoot outs" with some objective party refereeing the competition.

Overview of Workshop Discussions

On the issue of what is the best design method for the class of systems under discussion, there was virtually no agreement. Those who discussed design methods all discussed different methods. Furthermore, some of those whose specialty is scheduling theory questioned whether the design methods presented were real-time design methods at all. Their objection is that time is not taken into consideration in the design methods in such a way that "schedulability analysis" can be conducted from the very beginning of the design process. Schedulability analysis provides the ability to determine if the individual tasks comprising a system will meet their deadlines.

On the other hand, several who have been engaged in real-time design projects were not concerned whether schedulability analysis could be performed from the beginning of the design process. Everyone thought that some form of schedulability analysis was eventually necessary (the extreme being to *validate system timing during* system testing). It was agreed during the discussions that most real system timing requirements are stated in terms of end-to-end, top level requirements rather than in terms of deadlines for individual tasks, which are hardly ever known at the time of system specification. Some designers argued that the chief concern during the initial phase of design was the structuring of the system to meet these top level requirements rather than worrying about individual task deadlines, since the individual tasks might not even be known so early.

But almost without exception, the scheduling theorists seemed to be of the opinion that the first thing that needed to be done was to decompose the systems into tasks in order that schedulability analysis could be performed, arguing that at all times there needs to be some level of assurance (less certain at first, more certain as the design progresses) that the set of tasks into which the system is decomposed at the present time, can meet their deadlines. All the real-time design methods discussed at the workshop involved some method of using system requirements to structure the design, often taking into consideration other features than simply the timing requirements. This leads to system decompositions (into tasks) that are often hard to analyze using current scheduling theory tools. Essentially, the designers want to be able to make design decisions not necessarily related to timing requirements and then have scheduling tools that can accommodate these designs, whereas, the scheduling theorist wants designers to work within a framework that makes the scheduling problem more tractable.

The end result is the previously mentioned "disconnect" between design theory and scheduling theory. The discussions revealed that this disconnect is very real and needs to be given careful consideration during the design of large, complex, real-time systems. For small single processor, real-time systems, e.g., intelligent controllers or low level sample data systems, a designer often can work with current scheduling theory. But it must be recognized that the assumptions of some of the theories, e.g., rate-monotonic scheduling theory, impose considerable restrictions on the design process (for instance, the assumptions of a single processor, non-distributed architecture; of periodic or possibly sporadic tasks; of static periodicities or perhaps a small finite collection of modes; of known task execution times, etc.), to the point of being unrealistic to many designers. At present, the prospect for developing scheduling theories that support designers, in the way they would like to design, is limited. There are some results in this area (see for instance the paper by Le Lann), however, generalized results (ones not predicated on highly restrictive assumptions) appear to be difficult to realize.

During the presentations and discussions of the second day, it was generally agreed that real-time design theory and real-time scheduling theory are related, but there was no agreement as to what the relationship should be. Designers usually view schedulability analysis as a tool (one of many) to be used in the design process. They are not necessarily convinced that even though you can apply some scheduling theory to the tasks comprising their design, and even though the theory predicts that the tasks are schedulable, that the real system will really behave this way when it is coded and tested. This is because of the many factors that affect system timing that are difficult to take into consideration when applying the theory, especially when distributed or parallel processing elements are present in the design. What designers want is a scheduling theory that takes into consideration the realities of the problem space they are designing in. They do not consider analyses based on restrictive assumptions particularly relevant to the design process. In fact, most of them are not sure what weight should be attached to such analyses.

On the other hand, scheduling theorists usually believe that the proper timing behavior is so critical in real-time systems (it is this factor that distinguishes them from other types systems), that the whole design process needs to be structured around schedulability analysis. For otherwise, one is likely to produce a system that is impossible to make meet its timing requirements. This marked difference in viewpoint will likely continue to prevent complementary design and scheduling theories in the near future.

During the third day, surprising consensus was achieved regarding the issue of validation of real-time systems. Essentially everyone was in agreement that in order to validate that real-time systems meet their timing requirements, and continue to meet their timing requirements when fielded, it is necessary to instrument the code with some type of event recording package, and that the instrumentation code remain a part of the fielded system. It was agreed that most code debuggers, with which anyone had experience, alter the timing of the system under test and are therefore of little value in validation. Further, it was agreed that the few general code instrumentation packages that exist, have too much overhead to be permanently left in the code. The current state of the practice is for developers to build in their own test instrumentation, using specific knowledge of the application to achieve efficiency.

This built in test instrumentation code also provides a "window" into the system during operation. It is constantly running, dumping event occurrence information into some area of memory which is overwritten when full. In the event of a system failure, the events that occurred for some time period in the past (depending on memory size) are preserved. It is only in this way that we can hope to reconstruct what happened. Without this technique, it is generally not known how to analyze timing failures in real-time systems since they may not be readily reproducible.

During the last day of the workshop, participants discussed both the issue of where resources should be placed to try to solve the design methodology and the design vs scheduling disconnect problems (since there was total agreement on the best way to validate real-time systems it was no longer treated as an issue). Two suggestions that received a good deal of support involved narrower aims; namely, determining the role of object-oriented technology in the real-time problem domain, and standardizing real-time Inter-Process Communication (IPC). Both of these suggestions are directed at pieces of the over-all design process that are becoming critical in current designs. Beyond this, it was recommended that there should be more funding support for experimentation with (as opposed to creating more) real-time design methodologies, and that this experimentation should seek to compare and quantify the value of these methodologies in various situations. Finally, there was the suggestion that received almost total support, that there be support (funding) for some software design "shoot outs" of one or more representative real-time system, and that some objective party referee the competition.

List of Workshop Position Papers

List of Attendees

1. Prof. Ashok Agrawala, University of Maryland
2. Dr. Jay S. Bayne, Bailey Controls
3. Mr. William H. Booth, Encore Computer Corporation
4. Prof. Alan Burns, York University
5. Mr. Ed Chevers, NASA Ames Research Center
6. Dr. Edward Feustel, Institute for Defense Analyses
7. Dr. Dennis W. Fife, Institute for Defense Analyses
8. Dr. Armen Gabrielian, Uniview Systems
9. Prof. Richard Gerber, University of Maryland
10. Mr. Andre Goforth, NASA Ames Research Center
11. Dr. Norman Howes, Institute for Defense Analyses
12. Dr. E. Douglas Jensen, Digital Equipment Corporation
13. Mr. Robert J. Knapper, Institute for Defense Analyses
14. Prof. Jane Liu, University of Illinois
15. Dr. C. Douglass Locke, IBM
16. Dr. Kjell Nielsen, Hughes Aircraft Company
17. Dr. Asghar I. Noor, Institute for Defense Analyses
18. Dr. Lui Sha, Software Engineering Institute
19. Prof. S. Son, University of Virginia
20. Mr. David A. Wheeler, Institute for Defense Analyses
21. Mr. Jonathan D. Wood, Institute for Defense Analyses

22. Dr. Wei Zhao, Texas A&M University

Tomorrow's Challenges for Real-Time Systems

Ashok K. Agrawala
Department of Computer Science
University of Maryland
College Park MD 20742
(301) 405-2665

Real-time control systems have been around for a long time in the form of controllers for closed, limited functionality systems such as those needed for the control of an appliance. Such controllers were implemented using analog devices and converted to digital implementations through a complete and detailed analysis of the closed, limited functionality environment supported by such a system. But in the real world there are many examples of systems: which have significant real-time requirements, which must support an open environment where all applications and their execution sequences are not known, or are so large that they can not be taken into account in the design, which have significant reliability and fault tolerance requirements.

Sometimes such systems have been referred to as large, mission-critical systems(LMCS). A typical example is the air traffic control system. Design, implementation and life-time support of such systems pose many new challenges to the designers and researchers in the field. These challenges are not only for hardware technology or scheduling. This problem requires an integrated approach of hardware, scheduling, fault tolerance, distributed operation in the form of system technology that can support the systems of tomorrow systems effectively.

A typical general purpose computing application is characterized by the lack of any timing constraints within which it must execute. On the other hand, real-time applications must execute within timing constraints which may be soft or hard. Hard constraints must not be violated while soft constraints may be violated with defined penalties. An LMCS usually has to support all three types of applications and must, therefore, provide means for the timely execution of hard, soft and non real-time applications while permitting controlled interactions among such applications.

Reliability and fault tolerant operation are very critical requirements for LMCS. Such systems must continue to meet the timely execution requirements even when some faults occur. Many of the techniques developed for handling faults do not support the timing requirements of a real-time system. The design of LMCS has to address the conflicting goals of fault handling and real-time operations such that both goals can be met. Further, the faults may occur not only because of the failure of a hardware or software component; they may be caused by other conditions, such as overload conditions resulting from inputs or signals received beyond the designed capacity of the system. It is essential that the LMCS continue to function under such conditions, supporting at least the critical applications, and have degraded modes of operations.

Due to the conflicting requirements of real-time operation and fault-tolerant operation it is essential that the fault tolerant aspects of the system be an integral part of the system design rather than a separate add-on capability.

In order to support the required functionality of operation from multiple locations and fault tolerance the LMCS has to be implemented as a distributed system. A distributed system which supports real-time as well as fault tolerance requirements poses many new challenges. Temporal coordination as well as the synchronization of the time at different machines has to be supported. The communication among machines has to be carried out within the defined time constraints and the resources a various sites have to be managed in a unified manner.

The LMCS require comprehensive solutions which can be implemented and supported during the life time of the system in an integrated manner. We believe that straight forward adaptation of the techniques developed to address the system design problems in isolation are not likely to yield such comprehensive solutions. The life cycle support requires that not only the typical CASE tools be available but many additional tools be available which permit the analysis of the resource management, fault handling and timing requirements for the integrated system.

The development of large systems require that the techniques used be scalable. Many techniques which work well for small problems either do not work for large problems or cause such overheads that they can not be used for real-time operation.

One example of a project aimed at addressing the issues outlined above is Project Maruti at the University of Maryland. The goal of this effort is to develop a machine independent system which supports LMCS. The current prototype design implements Maruti kernel on Mach with minimum changes to the Mach kernel. It supports the multiple time domains of applications and fault tolerance requirements from the lowest level to the application overload handling. Scheduling and resource management techniques developed in this effort support the distributed, fault tolerant operation of the system and are scalable. Tools are being developed to support all the phases of the life cycle of an application.

Position Paper
Prepared for the Institute for Defense Analysis

WORKSHOP ON LARGE, DISTRIBUTED, PARALLEL ARCHITECTURE REALTIME SYSTEMS

Jay S. Bayne, Phd

Vice President, Research & Development
Bailey Controls Company
29801 Euclid Avenue
Wickliffe, OH 44092
216.585.5501 (v), 216.944.1008 (f)

Abstract The interrelated subjects of the design and the validation of large scale, distributed, realtime systems are critically important to the high volume, global, commercial process control industry. A key requirement for these mission critical control systems is *scaleability* with respect to such attributes as functionality, predictable performance, degree of distribution, fault tolerance, and serviceability. The thesis of this paper is that scaleable commercial-grade realtime systems do not presently exist, even though the global process control market demands such systems. The requisite base technologies exist in various forms, some mature, some embryonic, but their synthesis into stable, reusable, platform products has not received the attention of commercial suppliers. This has retarded the acquisition of a body of experience and the development of associated tools to support the needed system design theories, specification and development environments, and verification and validation methodologies.

The invitation to the "Workshop on Large, Distributed, Parallel Architecture RealTime Systems" (WorLDPARTS ?) defines four major issues to which are offered the following positions. The context of this response is defined by the requirements of large scale "mission critical" plant and process control applications that are routinely found in the global continuous process manufacturing industries. These industries include chemical, petrochemical, mining and minerals, steel, pharmaceutical, electric utility, food and beverage, waste water treatment, and pulp and paper segments.

Plant and process control systems for these segments involve thousands of I/O points (Level 0 transducers), hundreds of regulatory control loops (Level 1 cell controls), tens of supervisory controls (Level 2 inter-cell, or area controls), one or more sets of plant level controls (Level 3 inter-area, or plantwide controls), and one or more sets of inter-plant controls (Level 4 enterprise controls). Mission critical control applications comprise those hardware/software systems that have primary responsibility for plant and process production-, safety-, quality-, and regulatory agency reporting-related automation.

Within this application domain, the notions of "large, distributed, parallel architecture, realtime systems" has very specific meaning. "Large" implies both wide physical distribution (e.g., a campus setting), multiple nodes or network "end-systems" (e.g., 20-100 computational elements), and/or logically or physically complex node configurations. "Distributed" connotes both that applications (i.e., execution threads) span nodes, and nodes are physically isolated, often geographically close to the manufacturing processes they are responsible for controlling. "Parallel architecture" implies computing elements that are either uni-processors that operate in n-for-one redundant sets, or multi-processors that operate in loosely- or tightly-coupled arrays with local and/or shared global memories (e.g., symmetric SISD or SIMD configurations). And finally "realtime" implies that command and control application tasks must operate within specific "hard" [real]time constraints that must be guaranteed as part of the "correct" operation of the system.

Although not mentioned in the invitation, we believe that a critically important attribute of such systems is "scaleability". Ideally, we would be able to allow functionality, performance, timeliness, predictability, decentralization, and fault tolerance to be scaleable attributes of distributed realtime systems. System elements should be able to scale upwards from small sets of command and control services to large sets with known cost and complexity measures. Scaleable services, if properly implemented, would provide a tangible basis for building "correct" systems from the ground up, realizing a commercially important ability to incrementally expand the system while preserving its basic correctness.

The commercial "distributed control system" (DCS) business celebrates its 15th birthday this year, and is now in its second or third generation of technology, beginning with centralized minicomputer-based data acquisition and control (SCADA) systems and culminating in networked, microprocessor-based distributed computing systems. The essential macro elements of today's systems include I/O subsystems, fault-tolerant process controllers and data servers, and high performance, graphics-oriented human interfaces providing command and control console functions. These elements typically operate over tens of kilometer distances at 1-10 Mbps on redundant fiber, twinax, or coax backbone networks governed largely by proprietary protocols. The base technologies used within these contemporary macro products is changing rapidly, and over the short term will look conservatively something like the following.

Circuit densities are increasing at about 25% per year, doubling every three years. Device speeds are increasing at a similar rate. This is equivalent to realizing the same device functionality in half the space at twice the speed every three years. As a related development, the cost per processor instruction cycle is declining at 25% per year. This yields 100% additional processing capacity (operating at twice the speed in half the space) for the same cost every three years. The basis today is 25 MHz machines. By the mid-life of a new system we will be able to use 200 MHz processors in the same physical space and at the same prices as today's machines.

The cost of memory is declining at 15% per year, dropping by a factor of two every five years. DRAM densities are increasing at about 60% per year, quadrupling every three years. Therefore, in the span of just 10 years we should see twelve times the memory density at one quarter the cost. At the same time, application address space is being consumed at one additional address bit per year, on average, suggesting we need an additional 10 bits of address over the design half-life of a new machine. In today's control systems we use about 17 bits of address space per Level 0 device, 21 bits per Level 1 device, 23 bits per Level 2 device, and 24 bits per Level 3 device. By the year 2005 we estimate that Level 0 devices will utilize 26 address bits, with 32 bits at Level 1, 34 bits at Level 2, and 36 bits at Level 3. Clearly, 64-bit processors are required to implement the upper domains of the next generation of machines.

Disk density is increasing about 25% per year, doubling in three years. This keeps pace with the consumption of DRAM, and suggests that over the life of the system secondary storage demands will increase for two principle reasons. First, backing storage is required to contain (at least part of) the static images of the Level 1 through Level 4 machinery. Second, significant archival storage is required to log the operating history of the plant. For example, a plant with 1,000 field measurements sampled at 1 Hz would produce a raw Level 0 data rate of 64 Kbps, assuming 64 bits per point (data, plus status, plus time stamp). That represents a potential uncompressed Level 0 storage requirement of over 2 Terabits per year, or 250 Mbytes per point per year. Assuming an average compression factor of .6, we can estimate an appetite of 150 Mbytes per point per year of required archival storage capacity.

Available communications bandwidth is increasing by a factor of 10 every three years. Its basis today is 10 Mbps, yielding 100 Mbps by 1995, and 10 Gbps by 2005. This bandwidth is expected to be absorbed for a number of reasons, primarily at automation Levels 2 and 3, including: i) the routine use of multimedia man-machine interfaces that support in:egrated voice and full-frame video display systems; and ii) the increasing utilization of optical sensors. These sensors have application in many control domains, but when used for high speed flat sheet production (such as steel, film and paper making) can produce enormous volumes of data in very short periods.

This brief summary suggests that by the end of the design half-life of the next generation of plant control systems (circa 2005) the computational elements will routinely operate at 200-400 MHz, support address spaces of 30-40 bits, intercommunicate at 1-10 Gigabits per second over optical paths, collectively track and control an evolving plant state comprising over 10^6 objects, and utilize Terabyte backing storage subsystems. This expectation points to the real system design problem -- software -- its creation, configuration, deployment and maintenance.

With these few remarks as background the *Issues*, as defined in the IDA invitation, are addressed in corresponding *Position* statements. These *Positions* are admittedly strategic and terse in nature. Technical details would get messy. It is anticipated that the following comments will provide sufficient material to begin more in depth discussions.

Issue 1 "What is the best method or methodology for designing large, distributed realtime systems where processing elements may have a parallel architecture?"

Position 1 First, distributed, realtime control systems that utilize computational elements based on parallel architectures also generally support non-parallel architecture elements within the same design. The realtime nature of such systems demands that the distributed threads of control carry with them scheduling semantics which guarantee predictable (i.e., timely) performance across both classes of elements. Therefore, both the parallel and non-parallel architecture elements of the distributed system must support the same *virtual-machine* (i.e., VM or kernel) services that provide for the realtime "transnode" threads. This requires an underlying realtime IPC mechanism that supports both the parallel-architecture (i.e., tightly coupled, shared memory environment) elements as well as the non-parallel architecture (i.e., loosely coupled, message-based) elements of the distributed system.

Second, to support the development of *scaleable* end-use applications that implement the mission critical control policies of the system, the applications must be modular, and implemented in such a manner that they may be bound (compiled, linked, and/or interpreted) onto one or more processing elements of the distributed system. This suggests a set of formal methods are required that are inherently "object-oriented" in nature, at least in specification and design, if not in implementation.

These and other cogent reasons suggest two important rules for designing large, distributed, realtime systems: i) separate policies required for system coordination and management from mechanisms used to implement them within and across various system elements, and ii) clearly separate the mission critical applications from the underlying virtual machine by implementing formal application program interfaces (API's) which enforce the system design rules.

The separation of system coordination and management policies and mechanisms allows for the "objectification" of the underlying system elements while allowing applications to implement for themselves selected policies. This partitioning supports the (potentially dynamic) modification of policies as the system runs, providing for adaptation and reconfiguration of control regimes as external conditions and/or internal system faults warrant. It also allows for different mechanisms to be used by different system elements to enforce the same system policies.

The API's provide for stable interprocess (inter-object) semantics from which verification and validation can proceed. The interface(s) may promote the memory or processor models of underlying computing elements (i.e., shared global address space vs. local memories holding message-based agents), depending on the needs of the computations. It is my contention that with the speed of processing elements, the size of available memory subsystems, and the speed of interconnect networks, the justification for shared memory is arguable. A fundamental design issue for scaleable, distributed, realtime systems is location transparency. The means to achieve it are policy and mechanism issues.

The parallel-architecture elements of the distributed system may well require a shared address space to carry out their local computations. With the availability of 64-bit microprocessors (e.g., DEC's Alpha_AXP) and high speed mesh interconnect structures (e.g., CHPC's GalacticaNet) there are a number of proposed parallel architectures that utilize a flat, global directly addressed memory. While of some academic interest, this design does not meet the needs of a scaleable, mission critical distributed control systems.

Issue 2 "What should be the relationship between realtime design theory and realtime scheduling theory in a design methodology for this class of systems?"

Position 2 Their are a number of concepts associated with this issue that are system platform (i.e., virtual machine) related, and a number which properly belong to the control application problem domain. The first point I would make here is that the two domains are different and must be clearly understood. Too often the issues of scheduling are considered to be within the domain of the host operating system when they more properly belong to the set of application domain objects responsible for some mission critical function. Again, the design issue here is the clear separation of mechanisms provided by the underlying VM in support of application-level policies required to complete some task or transaction in a timely manner.

The implication of this point of view is to make scheduling parameters part of the application object's capability list. The activation (i.e., invocation) of an object will then dynamically associate its scheduling parameters with the underlying VM services responsible for the thread(s) executing within the object's address space. A dual of this viewpoint is to associate scheduling parameters with a computation's (i.e., process or task) execution thread at the point of its creation. These parameters then "follow" the thread as it meanders through the distributed system in pursuit of named objects (e.g., agents or actors) whose services are required of the computation. In both cases the scheduling of the object (or the scheduling of the task thread within the object) is governed by the semantics of the application as opposed to the operating system of the particular element executing the object's methods.

Here the realtime design issue is split into two parts: i) the scheduling policies required by the application to meet its time requirements (e.g., deadline, highest priority, or best effort), and ii) the mechanisms to be provided by the underlying VM for enforcing a formal and predictable allocation of system resources for realizing the policies within a population of (potentially) competing threads.

Issue 3 "What is the best method for validating that large, distributed, parallel architecture realtime systems behave as specified?"

Position 3 This remains an open question. There are the classical, conservative methods of component, subsystem, and ensemble testing to define the execution profiles of the system's VM services and application objects under "typical" operating conditions. There is "scenario analysis" based on certain assumptions about the probabilities of system events (e.g., internal faults, external events), often based on Monte Carlo simulations and the like. And for sufficiently small systems, there are correctness proofs and/or exhaustive testing.

But, in general, the problem of designing a large, distributed, realtime system that can be tested and verified "correct" under all of the non-deterministic operating conditions it might face is difficult. In support of our efforts at designing high-volume, commercial-grade, distributed control systems that govern the behavior of hazardous (e.g., chemical, refining, power production, pharmaceutical) processes, this issue is receiving a great deal of attention.

Current systems achieve validation through 1) static configurations, 2) limited applications functionality, 3) conservative implementations, 4) and exhaustive testing. These techniques work relatively well for control of low level regulatory loop control problems. They will not be sufficient to handle the larger application domains envisioned for our next generation of plant control systems. In current systems, critical realtime applications reside completely within a single node (element) in the distributed system.

In the next generation, applications will span nodes and require "transnode" validation of performance.

The means to achieve predictable, correct performance of distributed applications is through strict adherence to encapsulation, reuse, and location transparency in design; and to implement system elements with clearly defined interfaces that guarantee that parameters crossing the boundary are consistent (e.g., type and range checking). Invocation side effects must be bounded. System messages, from asynchronous fine grain signals (e.g., interrupts) to synchronous coarse grain IPC messages (i.e. request-reply semantics), must be universally understood.

With these facilities in place, supported by appropriate directory (e.g., request broker) services, the distributed nature of the system may be understood by verification and validation test suites applied, incrementally and iteratively, to selected ensembles of system objects. These various subsets, whose interactions provide for key mission critical services of the distributed realtime environment, can be validated prior to their engagement in the larger population whose collective behavior is even less deterministic.

These are basic principles behind, and motivations for, object oriented systems. And they are well understood, in principle. What is missing are formal and consistent specification methods, object interface (message) semantics, and the associated tools and techniques for realizing the V&V functions expressed above. There are a number of candidate disciplines (e.g., IDEF-based), but they are not generally adequate for large control systems which exhibit asynchronous, non-deterministic behavior. They are better suited to large "transaction-oriented" systems whose request-reply semantics are well defined (e.g., banking ATM's).

Issue 4 "Given that resources were available to enhance the design and testing methodologies for this class of systems, what are the most promising areas where these resources could be applied?"

Position 4 The area where current technologies and practices are most deficient is in the specification and implementation of task completion-time constraints, especially when these computations are carried by transnode threads. The problem is difficult in a single node system, whether that node be uni- or multi-processor in construction. The problem of distributed realtime guarantees is especially difficult because the underlying VM's do not provide the realtime IPC (RPC) mechanisms that provide for the requisite services. CMU's Archon Project, and its Alpha OS, provides an excellent example of an implemented solution for this general problem, but is suffers from i) not being a commercial system, ii) not being supported by formal methods and tools, and iii) being *avant garde*.

The availability of commercial-grade facilities must wait for OSF's realtime extensions to its MACH microkernel, extensions to Chorus, next generations of Sun's Solaris, IBM's rationalization of OS/2 and AIX, and so on. These distributed "object-oriented operating systems" will likely see commercial application in the late 90's. Their introduction will encourage the development of tools, object libraries with standard interface semantics (e.g., CORBA, DOMF), and a body of experience from which to develop formal methods.

The most promising areas for support are i) the specification and development of standardized realtime transnode IPC services which can carry scheduling information, ii) policies and mechanisms to provide for aggressively best effort scheduling of ensembles of threads executing on a given (uni- or multi-processor) node within a distributed system, and iii) methods and tools for expressing (in the functional requirements, design, and V&V documentation) precisely the completion-time performance required of a distributed computation. This includes he effects on the external environment and the internal state of the distributed system of being early, on time, or late.

Large, Distributed, Parallel Architectures for Real-Time Systems

POSITION PAPER

A. Burns
A.J. Wellings

Department of Computer Science,
University of York, UK

1. INTRODUCTION

The following material considers the four issues raised in the Invitation to Attend:

- What is the best method or methodology for designing large, distributed real-time systems where processing elements may be parallel architectures?
- What should be the relationship between real-time Design Theory and real-time Scheduling Theory in a design methodology for this class of system?
- What is the best method for validating that large, distributed, parallel architecture real-time systems behave as specified?
- Given that resources were available to enhance the design and testing methodologies for this class of system, what are the promising areas where these resources could be applied?

In giving a more detailed response to the first question a number of the other issues are addressed. We focus, in this position paper, on non-functional issues such as timeliness and dependability.

We assume that the allocation of software objects to the set of distributed nodes is essentially static (i.e. is undertaken before execution and then remains unchanged unless reconfiguration following significant failure is employed). Within a node, where parallel architectures may be employed, allocation is dynamic. We also assume that the distributed nodes are linked by shared communication media.

2. DESIGN METHODS

2.1. A Design Framework

The most important stage in the development of any real-time system is the generation of a consistent design that satisfies an authoritative specification of requirements. Where real-time systems differ from the traditional data processing system is that they are constrained by certain non-functional requirements (e.g. dependability and timing). Typically the standard structured design methods do not cater well with these types of constraints[8].

It is increasingly recognised that the role and importance of these non-functional requirements in the development of complex critical applications has hitherto been inadequately appreciated[3]. Specifically, it has been common practice for system developers, and the methods they use, to concentrate primarily on functionality and to consider non-functional requirements comparatively late in the development process. Experience shows that this approach fails to produce dependable real-time systems. For example, often timing requirements are viewed simply in terms of the performance of the completed system. Failure to meet the required performance often results in ad hoc changes to the system. This is not a cost effe‸ti·e process.

If hard real-time systems are to be engineered to high levels of dependability, the real-time design method must provide:

- the explicit recognition of the types of activities/objects that are found in hard real-time systems (i.e. cyclic and sporadic activities);
- the explicit definition of the application timing requirements for each object;
- the explicit definition of the application reliability requirements for each object;
- the definition of the relative importance (criticality) of each object to the successful functioning of the application;
- the support for different modes of operation — many systems have different modes of operation (e.g., take-off, cruising, and landing for an aircraft); all the timing and importance characteristics will therefore need to be specified on a per mode basis;
- the explicit definition and use of resource control objects;
- the decomposition to a software architecture that is amenable to processor allocation, schedulability and timing analysis;
- facilities and tools to allow the schedulability analysis to influence the design as early as possible in the overall design process;
- restriction on the use of the implementation language so that worst case execution time analysis can be carried out;
- tools to perform the worst case execution time and schedulability analysis.

A constructive way of describing the process of system design is as a progression of increasingly specific *commitments*[7,2]. These commitments define properties of the system design which designers operating at a more detailed level are not at liberty to change. Those aspects of a design to which no commitment is made at some particular level in the design hierarchy are effectively the subject of *obligations* that lower levels of design must address. Early in design there may already be commitments to the structure of a system, in terms of object definitions and relationships. However, the detailed behaviour of the defined objects remains the subject of obligations which must be met during further design and implementation.

The process of refining a design — transforming obligations into commitments — is often subject to *constraints* imposed primarily by the execution environment. The execution environment is the set of hardware and software components (e.g. processors, task dispatchers, device drivers) on top of which the system is built. It may impose both resource constraints (e.g. processor speed, communication bandwidth) and constraints of mechanism (e.g. interrupt priorities, task dispatching, data locking). To the extent that the execution environment is immutable these constraints are fixed.

Obligations, commitments and constraints have an important influence on the architectural design of any application. We therefore define two activities within architectural design[3]:

- the logical architecture design activity;

- the physical architecture design activity.

The logical architecture embodies commitments which can be made independently of the constraints imposed by the execution environment, and is primarily aimed at satisfying the functional requirements. The physical architecture takes these and other constraints into account, and embraces the non-functional requirements. The physical architecture forms the basis for asserting that the application's non-functional requirements will be met once the detailed design and implementation have taken place. It addresses timing and dependability requirements, and the necessary schedulability analysis that will ensure (guarantee) that the system once built will function correctly in both the value and time domains (within some failure hypotheses).

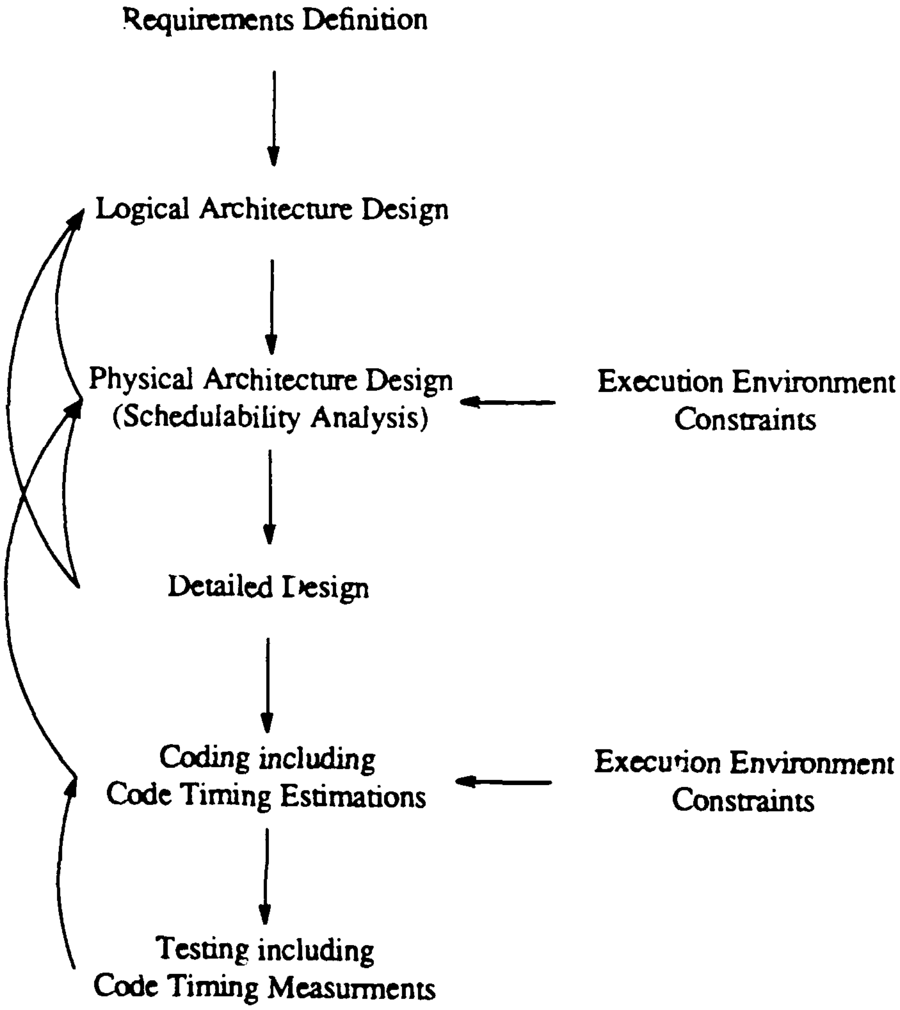


Figure 1: The Hard Real-time Life Cycle

Figure 1 gives an overview of the proposed framework. It is important to note, however, that this figure does not identify phases (in a traditional waterfall model) but (potentially concurrent) stages (activities). The output of each stage is a "product" that can be independently evaluated. Consistency of notation between products clearly improved the design process (see the following discussions on computational models and HRT-HOOD).

2.1.1. Logical Architectural Design

There are two aspects of any design method which facilitate the logical architecture design of hard real-time systems. Firstly, explicit support must be given to the abstractions that are typically required by hard real-time system designers. *We take the view that if designs are to be well structured so that they can be analysed, then it is better to provide specific design guidelines rather then general design abstractions.* For example, supporting the abstraction of a periodic activity allows the structure of that activity to be visible to the design process which, in turn, facilitates its analysis. In contrast, allowing the designer to construct periodic activities out of some more primitive "task" activity produces designs which are more difficult to analyse. Clearly, care must be taken so that the design method does not become cluttered with too many abstractions but an adequate level of support is desirable.

The second aspect involves constraining the logical architecture so that it can be allocated to a distributed system and analysed during the physical architecture design activity. One means of achieving this is to choose an appropriate computational model. Concurrency is obviously an important abstraction within any such model. We do not, however, believe that a synchronous communication model, such as that contained in CSP or CCS, is the correct abstraction for real-time systems. Rather we support an asynchronous model in which active objects (i.e. object that can give rise to spontaneous computation) interact via asynchronous messages or non-active objects (which may be either entirely passive or provide some form of protection over the data being communicated between the active objects; e.g. mutual exclusion).

This computational model is used in the Mascot-3 design method[12], and the formal method TAM[11]. We have also embodied the model into the HRT-HOOD design method, see section 2.1.3.

2.1.2. Physical Architectural Design

The primary focus of this activity is the allocation of objects to the distributed system and the analysis of the worst case response times for transactions (precedence related objects) running through the distributed system. To achieve this the logical architecture must identify the run-time objects of the application and give estimates of their resource needs (CPU cycles, communication load etc). The available resources in the execution environment must obviously also be known.

Rather than design to budget we believe that top level designs should be analysed to obtain an estimate of their likely resource needs. These estimates should form the basis of an initial schedulability analysis. If the design will not fit then extra resources must be found or the scope of the design reduced. As detailed design and coding is undertaken the schedulability analysis is re-done. Fixed budgeting and early allocation (of computing activities to rules of the distributed system) reduces the flexibility required in the design process.

In order to get sufficient flexibility into the allocation process it must be possible for a design to be mapped on to the distributed system in many difference ways. Ideally it should not be necessary to artificially split coherent objects in order to facilitate distribution. Rather the design should articulate a population of objects that can be combined in many different ways.

An application expressed in the computational model described earlier can easily be distributed as the active objects only have an asynchronous relationship with one another. For example a cyclic object on one node can release (and pass data to) a sporadic object on another node by using a distributed implementation of the appropriate non-active object or asynchronous message. All non-active objects can be distributed without affecting their functionality. Only asynchronous messages are needed at the communication layer. The temporal behaviour of the distributed system will change (when compared with the non-distributed implementation), but this can be analysed by the scheduling model.

2.1.3. HRT-HOOD

Within a project funded by the European Space Agency (ESA) we have been developing a structured design method for hard real-time systems. We took HOOD (Hierarchical Object Oriented Design)[1] as a basis but modified the method in line with the framework outlined above. In HRT-HOOD (Hard Real-Time HOOD) there are five object types: PASSIVE, ACTIVE, PROTECTED, CYCLIC and SPORADIC. The CYCLIC objects execute periodically; the SPORADIC ones have a minimum inter-arrival rate; the PASSIVE objects offer no protection on the data they encapsulate whereas the PROTECTED objects can offer protection (but are non-active). ACTIVE objects have no inherent (defined) structure. In a real-time system they must decompose into terminal objects of the other four types.

There are strict rules as to how object types may decompose and use the interfaces of other object types. The culmination of the logical design activity is a set of terminal objects that are well defined (ie not ACTIVE). All CYCLIC and SPORADIC objects interact via PASSIVE or PROTECTED objects, or via explicit asynchronous message passing (ie all communication is asynchronous). For example a CYCLIC object may asynchronously release a SPORADIC object. If an immediate response from an object is required then an asynchronous message can trigger a transfer of control operation in a CYCLIC or SPORADIC object (if such an operation is defined in the object's interface).

The timing requirements of the design are represented as attributes of the objects.

Mapping from HRT-HOOD to Ada 9X and to Ada 83 (with task optimisation) have been undertaken[4,5]. Preemptive priority based dispatching is used and PROTECTED objects employ ceiling priorities to obtain their protection. Each CYCLIC and SPORADIC object contains a single thread (task). A case study implementation of a single processor system has been undertaken[6]. This involved re-engineering the Olympus satellite's AOBS (Attitude and Orbital Control System). The system was designed in HRT_HOOD and implemented using a modified version of Ada83 (to reflect Ada9X facilities).

HRT-HOOD is not presented as the definitive means of designing real-time systems. But it has been defined to directly address the list of issues given earlier. We believe these issues are crucially important.

2.2. Designing for Parallel Architectures

Whereas the level of concurrency in a distributed system is often explicit, the exploitation of parallel hardware requires an implicit approach. This is because the task of designing software that can execute, efficiently, with different levels of parallelism (including none) is exceedingly problematic. The design process therefore needs:

- Programming abstractions that can reduce the burden of constructing parallel programs.
- Operating system (kernel) that give support for "infinite" parallelism.
- Operating system (kernel) support for dynamic reconfiguration following the loss of computing elements.

All of these activities need also to be coordinated with an effective approach to real-time.

One possible approach here is to allow an object to define its operation in such a way as to allow a variable number of parallel threads to be created by the kernel. A minimum number would however have to be guaranteed for the worst case response time to be calculated.

3. DESIGN METHODS AND SCHEDULING THEORY

As indicated above the three key issues are:

(a) The rules of decomposition allowed in the design method must not compromise the need to analyse the complete design.

(b) Abstractions used in real-time systems (e.g. cyclic activities, deadlines, response times) must be supported directly in the design method.

(c) Scheduling analysis must be applied as early as possible (to an incomplete design).

To balance flexibility, predictability and efficiency, preemptive (or deferred preemptive) dispatching of priority assigned threads is recommended. Priorities being, essentially, static. The use of cyclic executives is considered to be too restrictive; the paper by Locke[10] argues this case convincingly. Rate monotonic scheduling analysis (RMSA) has been very successful in showing how an engineering approach can be applied to give predictions for the timing behaviour of application using priority based run-time dispatching. The standard equations in RMSA are exceedingly simple and use only a measure of processor utilisation to give predictions. More comprehensive analysis can be achieved by considering the worst case response times (WCRT) of the threads incorporated in the design. Recent analysis has used this approach to give predictable WCRT in the presence of:

- Context switches
- Release jitter
- Clock interruptions with manipulations of delay queues
- Threads with tight requirements for I/O jitter control
- Mode changes
- Sporadic threads that have deadlines unrelated to their worst case arrival rate
- Sporadic threads that arrive in bursts
- Deadlines less than, or greater than, period (for cyclic threads)
- Threads with more than one deadline
- Threads with offset relations (with respect to one another)

There remains work to be done with multi-processor (parallel) nodes (i.e. to make sure that this analysis is applicable to a group of parallel processors sharing the same set of threads).

The scheduling of the communications media is more problematic. TDMA is very inflexible; dynamic collision based ethernet protocols although theoretically predictable[9] cannot yet be recommended. A software token bus protocol would seem the best compromise. It can be analysed and can be integrated with the allocation and node scheduling activities. If fault-tolerance is also to be addressed then some form of atomic broadcast will be needed. This will also have consequences for scheduling.

It seems increasingly likely that DSGM (distributed shared global memory) will play an increasingly important role in distributed real-time systems. The performance and applicability of this technology makes it a genuine alternative to LAN based approaches. Although, in some ways, such memories can be simply modelled as *slow* ordinary memory this may not be an adequate model. More realistic scheduling analysis may be needed.

It is well known that predictions based on a worst case scenario are pessimistic due to sporadic activities not occurring as often as worst case and hardware performing better than can be relied upon (due to cache and pipelining). The incorporation of "unbounded" components into a design (so that they can make use of spare capacity) is currently an active research topic. Once paradigms are developed then the design methods must allow these new abstractions to be used directly. We anticipate new object types being added to HRT-HOOD.

Different scheduling regimes will impose different restrictions on the design process. But it must be the case that the needs of the scheduling theory dictate what the properties of the design method should be.

To facilitate flexible (static) allocation of objects to nodes it is important that objects in the design method are not closely coupled. The use of asynchronous thread interaction does give considerable freedom to the allocation process. But for large systems the complexity of the allocation and configuration activity remains significant. We have had some success at applying simulated annealing to the allocation activity[13]. Clearly tool support is needed.

4. VALIDATION

Formally proving large concurrent systems is beyond current engineering practice. Testing of highly dependable software cannot provide the reliability levels needed. It is therefore clear that the design method (and process) must itself support validation. The following points will help bring this about:

(a) Use formal methods to prove the sequential behaviour of objects.

(b) Use asynchronous interactions between threads.

(c) Use scheduling analysis to prove that the end-to-end timing requirements are met.

(d) Use proven (certified) run-time kernels.

The use of preemptive priority based scheduling should allow kernels to be certified.

5. PROMISING AREAS OF RESEARCH

It would seem unlikely that funding research in design methods themselves would be effective. Methods need good tool support and a user population to evaluate them. Rather it will be more cost effective to focus on particular aspects of the problem. We would give top priority to consideration of how an object (using the term to imply any module structure) interface and specification can be analysed to give a meaningful prediction of the resource needs of that object when it is implemented in software. Clearly this issue is linked to that of reusability and classification; it will also imply that the notation used to specify the interface and specification of the object will need particular expressive power.

Much of the necessary scheduling theory is in place although the ramifications of parallel architectures requires further study, and the scheduling of the communication media remains an open issue. But exemplar systems are still rare. The funding of case studies (i.e. implementations), and where appropriate tools, will help demonstrate the power of these techniques.

6. OTHER ISSUES

It is often the case that the requirement for *fault tolerance* is added to the list: *large, distributed, parallel* and *real-time*. If this is the case then a number of other issues need to be considered:

- The degree of parallelism may decrease over time (for long life non-stop systems). The application programmer should not have to program this reconfiguration.
- Communications must be replicated; either by acknowledgement and rebroadcast (if acknowledgement times out), or by diffusion (sending the message more than once to start with).
- Processing elements may need to be replicated. Within a node is easier but does not give the same fault coverage as replication between nodes. The latter, however, requires some form of atomic broadcast on the communication media.

Although all these issues have been addressed in non-real-time systems, the added requirement for timely performance significantly complicates the problems involved.

7. CONCLUSIONS

The needs for flexible allocation (configuration) and predictable performance necessitate the use of an asynchronous computational model. On top of this the design method should provide object types that correspond to the abstraction found in real-time applications. We, in HRT-HOOD, have used two forms on active object (CYCLIC and SPORADIC) and two forms of non-active object (PASSIVE and PROTECTED). All communications between active objects can be remote without effecting the functionality of the code.

Preemptive priority based scheduling (on the multiprocessor nodes), with an equivalent behaviour on the communication media, gives an appropriate level of flexibility for the allocation process. It also facilitates analysis of the timing behaviour of the application.

References

1. European Space Agency, "HOOD Reference Manual Issue 3.0", WME/89-173/JB (September 1989).

2. A. Burns and A.M. Lister, "An Architectural Framework for Timely and Reliable Distributed Information Systems(TARDIS): Description and Case Study", YCS.140, Department of Computer Science, University of York (1990).

3. A. Burns and A.M. Lister, "A Framework for Building Dependable Systems", *Computer Journal* 34(2), pp. 173-181 (1991).

4. A. Burns and A.J. Wellings, *Hard Real-time HOOD: A Design Method for Hard Real-time Ada 9X Systems*, Towards Ada 9X, Proceedings of 1991 Ada UK International Conference, IOS Press (1992).

5. A. Burns and A.J. Wellings, "Designing Hard Real-time Systems", pp. 116-127 in *Ada: Moving Towards 2000, Proceedings of the 11th Ada-Europe Conference, Lecture Notes in Computer Science Vol 603*, Springer-Verlag (1992).

6. C.M.Bailey, A. Burns, E. Fyfe and A.J. Wellings, "Implementing Real-time Systems: A Case Study", *Proceedings CNES Symposium Real-Time Embedded Processing for Space Application* (1992).

7. J.E. Dobson and J.A. McDermid, "An Investigation into Modelling and Categorisation of Non-Functional Requirements", YCS.141, Department of Computer Science, University of York (1990).

8. H. Kopetz, R. Zainlinger, G. Fohler, H. Kantz, P. Puschner and W. Schutz, "The Design of Real-Time Systems: From Specification to Implementation and Verification", *Software Engineering Journal* 6(3), pp. 72-82, Softw. Eng. J. (UK) (May 1991).

9. G. Le Lann, "The 802.3D Protocol: A Variation on the IEEE 802.3 Standard for Real-Time LANs", Internal Report, INRIA, France (1987).

10. C.D. Locke, "Software architecture for hard real-time applications: cyclic executives vs. fixed priority executives", *Real-Time Systems* 4(1), pp. 37-53, Real-Time Syst. (Netherlands) (March 1992).

11. D. Scholefield and H. Zedan, "TAM : Temporal Agent Model for Real-Time Distributed Systems.", in *Proc. EUROMICRO '90 - Hardware and Software in Systems Engineering, Sixteenth Symposium on Microprocessing and Microprogramming*, ed. D. Fay, Elsevier Science Publishers B.V. (1990).

12. H.R. Simpson, "The Mascot Method", *IEE Software Engineering Journal* 1(3), pp. 103-120 (1986).

13. K. Tindell, A. Burns and A. Wellings, "Allocating Real-Time Tasks (An NP-Hard Problem made Easy)", *Real Time Systems* 4(2), pp. 145-165 (June 1992).

The Role of Formal Methods in the Design of Complex Real-Time Systems*

Armen Gabrielian
UniView Systems
1192 Elena Privada, Mountain View, CA 94040
(415) 968-3476
armen@well.sf.ca.us

The design of complex real-time systems which may involve parallel architectures is a difficult task with no clear-cut solutions. The scientific analysis of the critical issues in this field, however, can only be realized by the availability of a formal framework for specification of system behavior and requirements, that abstracts irrelevant details and permits formal analysis. In particular, an executable specification can be considered as a "causal model" of a system that can be used for verification, as well as, for testing, fault diagnosis, analysis of scheduling requirements and performance evaluation. Analysis can reduce significantly the time and cost of evaluating complex parallel systems compared to the purely experimental approaches. Experimental efforts are still required, however, to validate the results of the formal analysis and to help determine the most promising research issues. Since requirements on systems can vary enormously, no single design methodology is guaranteed to be optimal. A formal framework in which both *requirements* and *system design* can be specified precisely is the key to reducing the development cost and verifying the adequacy of complex designs. The goal for optimality in this view is replaced by the goal of satisfying a set of pre-specified requirements. An overview of a particular specification method that meets many of these requirements is presented in this paper.

1 Introduction

In this paper, we address the following four questions raised in the Call for Papers:

1. What is the best method or methodology for designing large, distributed real-time systems, where processing elements may have a parallel architecture?

2. What should be the relationship between real-time Design Theory and real-time Scheduling Theory in a design methodology for this class of systems?

3. What is the best method for validating that large, distributed, parallel architecture real-time systems behave as specified?

4. Given that resources were available to enhance the design and testing methodologies for this class of systems, what are the most promising areas where these resources could be applied?

These questions can be addressed from various perspectives. Our approach is to explore the role that "formal methods" can play in the design of complex and safety-critical real-time systems. As indicated in [LL92], one can no longer rely on "engineering judgment" to assure safety and other properties of a design. Also, as predicted in [CK91], in the future, "programmers, tired of debugging difficulties, will use design methods that produce correct programs as well as formal methods of proving correctness and methods of integrating proofs and testing." Our goal is to

*This work was supported in part by the Office of Naval Research under contract N00014-92-C0047.

indicate how formal methods can contribute to the establishment of a scientific basis for designing complex real-time systems. In particular, our contention is that to be able to predict whether a design is adequate, one must have specifications of the problem and the design. These specifications must be formal enough to permit analysis of the properties of interest.

A formal method is defined here as "a formalism for specifying the properties of a system, in a way that formal analysis can be performed on it." We adopt a somewhat liberal definition for the term "formal" in this paper. However, we do not consider simulation, by itself, as a sufficient criterion for judging a specification formalism to be formal. On the other hand, not all formal methods provide a simulation capability (or executability). The most promising methods are those that are both executable and also offer a framework for mathematical analysis of behavioral or functional properties.

The potential benefits of formal methods are in (1) understanding the requirements and architectural implications of systems, and (2) in responding to the first three basic question raised above. Our basic tenet is that by using a formal framework, one can perform the kinds of analysis that are necessary to evaluate a design and, at the same time, address performance issues regardless of the types of architectures employed. Considering the difficulties often encountered in performance-oriented experimentation on actual parallel architectures [H91], the use of formal methods can reduce the cost of designing systems that must meet hard deadlines and strict performance requirements. Experimental work is still required, however, to support the formal analysis, particularly in validating assumption relating to communication delays and execution times of individual processes.

In the next section, we review the basic concepts of formal methods for real-time systems. In particular, we present a brief outline of the "hierarchical multi-state (HMS) machine" approach to specification and verification of systems that may involve parallel architectures. In Section 3, we discuss the benefits of formal methods in addressing questions relating to validation, performance, scheduling, etc. Finally, in Section 4, we present some conclusions and our recommendations for the most promising areas that require further expenditure of resources.

2 Overview of Formal Methods for Real-Time Systems

Numerous formal methods for non-real-time systems have been reported in the literature. The International Standards Organization (ISO) has accepted SDL, Estelle, and LOTOS as "standard formal description techniques." Process algebra, Z and Petri nets are some of the other techniques that have achieved some prominence, particularly in Europe.

For real-time systems, the main drawback of these methods is the absence of a natural representation for time. Numerous extensions, however, have been proposed that address this issue. For example, [NS92] deals with the temporal extension of process algebra and various timed extensions of Petri nets have appeared. Another major trend has been to modify temporal logic to deal with hard deadlines or to add delays in finite-state machine models of processes. A temporal logic perspective on these issues can be found in [MP92].

The principal problem with most of the formal methods for real-time systems is the inability to deal with complexity. For example, state proliferation makes the use of finite-state machine models difficult, except for relatively simple cases. The use of multiple communication finite-state machines limits this problem to some extent, but not sufficiently. The most promising state representations that address the complexity issue are hierarchical "multi-state" models, in which multiple states may be active at a particular level of hierarchy. One example is Statecharts [Ha87] which is used mainly for simulation; another is the HMS machine model to be discussed later in this section which was originally introduced in a simpler form in [GS87].

A combination of state modeling within a temporal framework seems to be the most promising way of specifying real-time systems. The standard way of accomplishing this, even for hierarchical state models, has been to add delays on transitions. This is a highly limited view of time since the language of delays is simply inadequate for dealing with complex temporal relationships. In contrast, pure temporal logic, by itself, is not capable of expressing all "regular properties" that are definable in terms of state models.

A more fundamental problem with all standard specification languages for real-time systems is the almost exclusive use of "future time" as a basis for defining behavior. Thus, in many formalisms, individual action are defined in the following form: "if the system is in state s, then if the event a occurs at time t, the system will move to state s' sometime between $t+t_1$ and $t+t_2$." The inherent assumption here is that the world is deterministic and *the future is predictable*. In point of fact, one of the basic characteristics of a real-time system is that its inputs are *unpredictable*. To build a "causal model" of a system for analysis, the only reasonable assumption is to define what will happen *now* or, in case of a discrete model of time, what will happen *at the next moment*. Consequently, it appears more natural to use a "past time" temporal language to define a system's behavior in terms of its history.

The "hierarchical multi-state (HMS) machine" methodology [GF88,Ga91a,GI92] combines state modeling approach with an interval-based temporal logic to give a comprehensive behavioral model of complex real-time systems that (1) reduces state complexity compared to traditional state models, (2) provides a rich language for expressing complex temporal relationships, (3) avoids assumptions of determinism of future events, and (4) provides executability, as well as the ability to verify properties such as safety and schedulability. An expressive graphic notation makes the formalism accessible to a larger audience than many other formal methods and a simple extension makes possible the specification of *unbounded* parallelism under temporal constraints.

Informally, an HMS machine H is a triple (S, Γ_D, Γ_N), where S is a set of "states," Γ_D is a set of "deterministic transitions," and Γ_N is a set of "nondeterministic transitions." Each state in S is either a "primitive state" or is an HMS machine itself that may be equivalent to H, in the latter case giving rise to a recursive hierarchy. Each primitive state is either TRUE or FALSE, thus extending the standard notion of state in systems theory to a system with possibly multiple true states. Each transition in Γ_D or Γ_N is a mapping from one subset of states of S (the "primaries") to another subset of S (the "consequents"). The "firing" of a transition at a moment of time causes its consequent states to become TRUE, while each primary state becomes false unless it is the consequent of a transition that fires simultaneously. In contrast to traditional automata theory, the enablement condition of a transition is defined not by messages but in terms of an interval-based temporal logic, called TIL. The logic TIL extends propositional logic by the addition of the several new operators. The main operators are: (1) $O(t)$ = *at* time t, (2) $[t_1, t_2]$ = *always* from t_1 to t_2, and (3) $<t_1, t_2>$ = *sometime* from t_1 to t_2.

The behavior of a real-time system can be defined in terms of an HMS machine by first defining its attributes hierarchically in terms of states. Secondly, the changes in its states are characterized by transitions. Thirdly, the conditions under which changes occur are specified by TIL predicates. Consider a process *Proc-A* that executes for a minimum of *5* seconds as long as an error flag is not raised and aborts immediately if the error flag is raised. This would be defined in terms two transitions out of the state *Proc-A* as in Figure 1. States are denoted by boxes, transitions by dark arrows and predicates by thin arrows using VLSI symbols for boolean operators and an encircled T for temporal operators. The vertical transition (representing normal termination) is nondeterministic (labeled with an asterisk) and has the TIL predicate *[-5,0]Proc-A* $\wedge$ $\neg$ *ERROR*. The horizontal transition (representing abnormal termination) is deterministic and has the predicate *ERROR*. At any moment of time, one can evaluate the predicates to determine if either transition must or may fire. The predicates essentially define the *causes* or *permissions* for the transitions to fire. For example, if *Proc-A* has been active continuously for at least *5* units of time, then the predicate *[-5,0]Proc-A* will be true and the process *Proc-A may* terminate if the state *ERROR* is not TRUE. Note that *[-5,0]Proc-A* will remain true if the transition does not fire immediately and no deterministic assumptions about the future are necessary. On the other hand, whenever the state *ERROR* becomes TRUE, the system aborts immediately. Under most traditional representations, one would have to *retract* the original statement that the process *A* is to terminate sometime after *5* units of time if the system aborts. In our notation no such retraction is required.

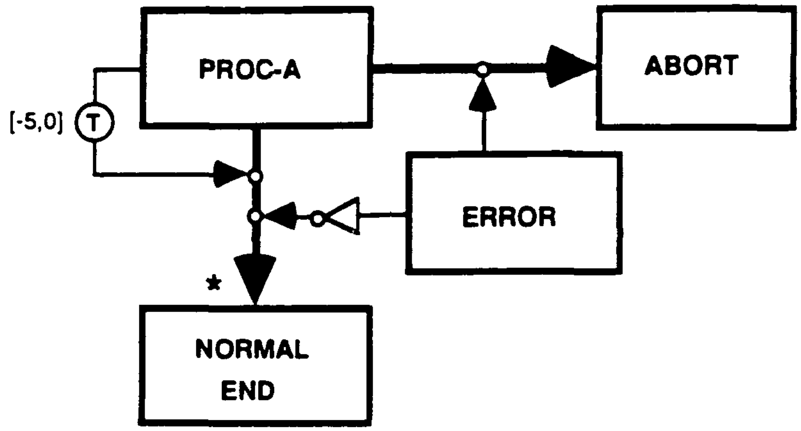


Figure 1. Specification of a Simple Real-Time Process

The key questions to be raised about any formal notation in the context of this workshop are: (1) Can it deal with complexity? (2) How does it address parallelism? (3) What benefits does it provide? The third question is addressed in the next section. We now turn to the first two questions.

To deal with complexity, abstraction mechanisms are needed that allow one to express and analyze relationships at higher levels without concern with lower-level details. Hierarchical decomposition is one common abstraction mechanism that many methods, including HMS machines, employ. Another mechanism in case of HMS machines is the use of multiple (partial) states to represent the traditional concept of the state of a system. This can potentially reduce the number of states of a systems logarithmically. For example, 2^N states in a finite-state machine may be required to specify a system described in terms of *N* states in an HMS machine. A third abstraction mechanism for HMS machines is nondeterminism. In traditional modeling methods such as Petri nets and finite-state machines, nondeterminism arises from structural considerations. For example, if two transitions from a state are triggered by the same signal, then the choice is made nondeterministically. In contrast, in an HMS machine, a transition is explicitly designated to be deterministic or nondeterministic. Nondeterminism in this form can be used to capture temporal uncertainty, specially early in the system design stage. It also serves as the key in relating specification to scheduling as discussed in the next section. Finally, a fourth abstraction concept for HMS machines is "multi-level specification," which was first introduced in [GF91]. In a multi-level specification of a real-time system, a hierarchy of specifications jointly describe the behavior of a system. The lowest levels normally consist of nondeterministic specifications that describe a whole class of behaviors. Upper-level specifications employ slightly more complex versions of HMS machines, called "policy machines," which describe goal-oriented behavior. The complete specification is obtained by finding paths or "plans" at the lowest level machines that satisfy higher-level constraints. As a result, a high degree of reusability and modularity of specifications is obtained. In an experiment in applying this approach to the specification of a distributed component of a command and control system [Ga91b], a significant reduction in the actual specification effort was obtained.

As far as addressing parallelism is concerned, most state-based approaches do not provide any convenient facilities. One must explicitly model the parallel processes individually and the degree of parallelism must be known *a priori*. Preliminary studies have shown that an extended version of HMS machines may provide a very general approach to dealing with *unbounded* parallelism. In the extended version, a state may contain an arbitrary number of "tokens," each representing one of a set of similar processes associated with the state. With a slight extension of the logic TIL, one can then obtain a powerful formalism for specifying and reasoning about parallel processes. In fact, this approach extends the "tagged-token" model of data flow machines proposed in [AG82,De86] by introducing conditionality and temporal dependencies in data flow operations.

3 Applications of Formal Methods

Given an executable formal specification of a real-time system, various types of analysis can be performed on it. In addition, limited success has been achieved in using a specification as a basis for synthesis of an actual implementation. For software, a common approach to synthesis has been "correctness-preserving transformations" that lead a specification gradually to a program in a desired language, the execution of which satisfies the requirements indicated in the specification. For a hardware subsystem, a specification may be transformed, after a set of transformations, into a hardware description language such as VHDL, from which gate-level logic diagrams may be constructed. In the remainder of this section, we will address the applications of formal methods in analysis of specifications.

Since an actual implementation of a system can be a costly and time-consuming process, an executable specification can be valuable in permitting simulation-based analysis to determine whether performance related requirements can be met. Historically, the critical problem in simulation has been the absence of accurate timing information. For air defense, air traffic control and other application systems, where historical information is available from similar previous systems, the problem is not too serious. For completely new systems such as SDI and new computer architectures or operating systems, the problem is much more acute.

An important application of a formal method is in its potential ability to verify formally properties of a system before it is actually built. For hard real-time systems, most of the interesting behavioral attributes can be grouped under "safety properties." A safety property is an invariant of a system expressible in the language of temporal logic as $\Box p$ (*always p*), where p is a "past" temporal logic predicate. Given an HMS machine specification of a system and such a safety property, one can always create a new "system failure (*SF*)" state such that the state *SF* becomes true if and only the safety property is violated. Figure 2 depicts an example, where the safety property is the deadline condition "*always B* within *20* time units of *A*." The vertical bar in the figure represents a state that always has the value TRUE and the transition fires if *A* was true *20* time units ago, while *B* has been FALSE for the last *20* time units.

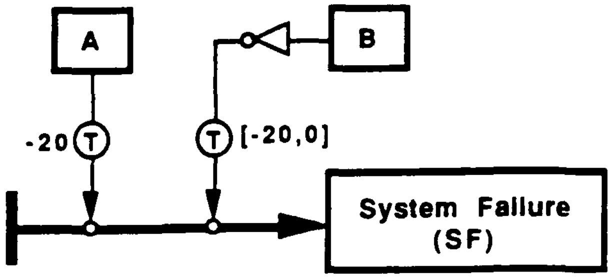


Figure 2. Representation of a Deadline Safety Property

Assuming that the states *A* and *B* are part of a larger specification, to verify the correctness of the specification with respect to the safety property, it is sufficient to demonstrate that the state *SF* is *unreachable*. In [GI92], a refutation-based theorem proving approach was presented for verifying such safety properties which is complete in the following sense: Given a safety property, if it is indeed satisfied by the specification, there exists a proof for it. The proof begins with the assumption that the safety property is violated and reasons backwards to demonstrate that the assumption leads to a contradiction in all possible realizations. The advantage of the method is that a complete enumeration of behavior is not necessary, in general. Thus, in principle, the proof of a simple property for even a very complex system could be quite simple. Theorem proving can also be performed in a forward reasoning form, once again, without the need for complete analysis of behavior. On the other hand, in an enumeration-based technique, even the verification of a simple property may require the creation of a complex computation graph. For certain types of properties, however, the enumeration-based methods are more suitable than theorem proving methods.

A specification can also be used for analysis of scheduling requirements of processes. In [GF91], a general method for deriving schedules for concurrent process from specifications was presented. In this scheme, one begins from the consideration of a nondeterministic HMS machine specification of a real-time system. Given a partially-ordered set of processes that must be executed, one can then derive a set of mathematical inequalities that must be solved in order to satisfy the local logical and temporal constraints. Such an approach can also be used for verification by demonstrating the infeasibility of schedules for reaching system failure states of the type indicated in Figure 2.

Finally, a formal specification may be used (1) to develop requirement-based test cases for evaluation of an implementation and (2) to perform diagnosis of causes of errors. For testing, a specification of the environment in which a systems is to operate will normally be required, while, for diagnosis, a backward reasoning process much like refutation-based verification can be used. No significantly new specification features are expected to be necessary in either case, although analytic and heuristic techniques are still needed to realize these goals in a realistic setting.

4 Discussion and Conclusions

For three reasons, it is our belief that the search for the *best* design methodology must be abandoned. First, the requirements of systems can be very different. Design method *A* may be far superior to deign method *B* for one application, whereas the opposite may be true for another application. Secondly, the implementation framework (architecture, operating system, communication facilities, and programming language constructs) may have a significant bearing on the quality of a design. Thus, for example, the type of parallelism, the synchronization constructs and the reliability of the communication systems can affect the performance of a system significantly. One design methodology cannot be expected to be satisfactory in all cases. Thirdly, new methodologies always arise that improve upon existing approaches. Thus, the justification for choosing a methodology even for a very restricted domain may be short-lived.

In the absence of a clear choice for a design methodology, the judicious approach seems to be encourage the development of tools and methods that make possible the rapid *evaluation* of a design under a set of requirements. Thus, formal methods for specification and analysis of systems that can deal with complexity, real-time issues, performance and correctness are recommended areas for further expenditure of resources. The search for optimality of design must also be given up since it is an impossible and unrealistic goal for large systems. The goal of design must be limited to meeting requirements under the specified constraints. Optimality is often undefinable and unattainable.

As far as validation of large systems is concerned, simulation and formal verification are the most promising approaches. The basis for both, however, should be a specification methodology that can lend itself to formal proof of correctness for critical aspects, can deal with systems at various levels of abstraction, and can avoid the abstruseness that is characteristic of many formal methods. The use of graphic notation is recommended to deal with the problem of accessibility of formal methods to a wider audience.

While there exist many verification techniques, it has been our experience that no single method is universally applicable. Thus, a variety of verification methods must be developed, some of which may be heuristic and incomplete. For a given system, the best choice of method cannot always be predicted. A strategy that has been successful in a different context has been to attempt several different techniques and experimentally determine the best approach for a given problem. In our experience, theorem proving either in the forward or reverse direction and scheduling-based analysis have proven to be the most promising formal verification methods. Also, certain enumeration techniques have shown great promise in hardware verification.

The use of a formal representation of a system can also provide a mechanism to investigate the relationships between design and scheduling theories. As stated in the previous section, it is possible to derive scheduling requirements of aperiodic and event-driven processes from a specification. There exists little experience, however, in actually deriving scheduling strategies from specifications for large systems.

Our final conclusions can be summarized as follows:

1. There exists no *best design methodology*. Different methodologies may be preferable under different requirements and implementation constraints. Also, rather than seeking an *optimal*

design, the aim should be to create a design that meets requirements. The cost associated with seeking an optimal design, even if it can be found, may not be worth the effort.

2. A framework is necessary to evaluate a design under an *arbitrary set of requirements* and *constraints*.

3. *Formal methods* for specification that can deal with complexity and provide capabilities for simulation, verification, automated support for testing and analysis of temporal properties can offer important benefits in the creation of large and distributed real-time systems.

4. *Experimental analysis* is needed to support and validate the results of formal techniques.

5. The *most efficient use of resources* will be in the development of integrated methodologies and tools for rapid evaluation and analysis of designs of complex real-time systems. In particular, formal methods can provide a common language for specification of requirements, simulation, verification, and automated support for testing. Small experimental laboratories are also necessary to validate the analytic methods. For the experimental work, the definition of a set of prototypical examples and benchmarks will be a necessary ingredient. Relatively modest expenditure of resources will be required if this approach is pursued, yet it could have a major impact on the creation of systems that utilize parallel and distributed architectures and meet operational requirement in a cost-effective manner.

References

[AG82] Arvind, and K.P. Gostelow, "The U-interpreter," *Computer* February 1982, pp. 42-49.

[CK91] Chandy, K.M., and C. Kesselman, "Parallel programming in 2001," *IEEE Software*, November 1991, pp. 11-20.

[De86] Dennis, J.B., "Data flow ideas and future of supercomputers," in *Frontiers of Super-Computing*. N. Metropolis, D.H. Sharp, W.J. Worlton and K.R. Ames (Editors), U. of Calif. Press, Berkeley, 1986, pp. 78-96.

[Ga91a] Gabrielian, A., "HMS machines: a unified framework for specification, verification and reasoning for real-time systems," in *Foundations of Real-Time Computing: Formal Specifications and Methods*, A.M. van Tilborg and G.M. Koob (Eds.), Kluwer Academic Publishers, Boston, 1991, pp. 139-166

[Ga91b] Gabrielian, A., "Multi-level specification of command and control systems," *Proc. Command and Control Research*. National Defense University, Washington, DC, June 1991, pp. 98-103.

[GF88] Gabrielian, A., and M.K. Franklin, "State-based specification of complex real-time systems," *Proceedings of the 9th Real-Time System Symposium*, Huntsville, AL, December 6-8, 1988, pp. 2-11.

[GF91] Gabrielian, A., and M.K. Franklin, "Multi-level specification of real-time systems," *Communications of the ACM*, Vol. 34, No. 5, May 1991, pp. 50-60.

[GI91] Gabrielian, A., and R. Iyer, "Verifying properties of HMS machine specifications of real-time systems," *Computer-Aided Verification*, Lecture Notes in Computer Science No. 575, K.G. Larsen and A. Skou (Eds.), Spring-Verlag, Berlin, 1992, pp. 421-431.

[GS87] Gabrielian, A., and M. Stickney, "Hierarchical representation of causal knowledge," *Proc. WESTEX-87 IEEE Expert Systems Conf.*, Anaheim, CA, June 1987, pp. 82-29.

[Ha87] Harel, D., "Statecharts: a visual formalism for complex systems," *Science of Computer Programming*, Vol. 8, No. 3, June 1987, pp. 231-274.

[Ha91] Howes, N.R., "Real-time Ada design methodologies and their impact on performance," Institute for Defense Analysis, Technical Report P-2488, Alexandria, VA, June 1991.

[LL92] Laprie, J.-C., and B. Littlewood, "Probabilistic assessment of safety-critical software: why and how?" *Communication of the ACM*, Vol. 35, No. 2, Feb. 1992, pp. 13-21.

[MP92] Manna, Z., and A. Pnueli, *The Temporal Logic of Reactive and Concurrent Systems - Specification*, Springer-Verlag, New York, 1992.

[NS92] Nicollin, X., and J. Sifakis, "An overview and synthesis of timed process algebras," *Computer-Aided Verification*, Lecture Notes in Computer Science No. 575, K.G. Larsen and A. Skou (Eds.), Spring-Verlag, Berlin, 1992, pp. 376-398.

Unifying Real-Time Design and Implementation

Richard Gerber
Department of Computer Science
University of Maryland
College Park, MD 20742
rich@cs.umd.edu

December 24, 1992

1 Introduction

The realization of a real-time system can often be an *ad hoc* process of experimentation. Many factors conspire to make this the case, among which are inflexible scheduling paradigms and the lack of high-level programming language support. Real-time performance is subsequently achieved by manually counting instruction-cycle times, hand-optimizing the code, and experimenting with various orderings of operations to help achieve schedulability.

This problem is compounded by the inherently iterative nature of system design. In the first step, mission goals are described in a rather informal fashion, and typically in a natural language. These informal goals are clarified in a requirements specification, and subsequently expanded into a system specification. This refinement process gradually proceeds until a final, complete implementation is constructed.

This design approach is replete with potential problems. For example, consider a system design D and its successor design D'. First, errors may be present in D, and then carried down to the more concrete design, D'. Second, errors may be introduced in D' that were not present in D and that may in turn become present in the final implementation.

These problems are especially prevalent in distributed real-time systems, where at each refinement stage, assumptions must be made about deadlines, scheduling algorithms, CPU speeds, clock drift, resource requirements, etc. Such design assumptions rarely hold in the full implementation, and thus, the real-time system will probably not meet its original mission requirements.

In theory, this process of stepwise refinement should be a natural one, and there should be a disciplined approach for detecting errors at each refinement stage. In practice, however, no such disciplined approach exists for designing large, real-time system. There is often a large schism between the design and eventual implementation of the system. At the design stage, implicit assumptions are made about the eventual implementation; for example, its number of resources, execution speeds, etc.

2 Formal Design and Scheduling

The area of formal methods offers several potential solutions to this problem. Yet while many formal models of real-time computation have been developed [1, 3, 7, 4, 10, 8, 13, 17, 19], most treat processes abstractly, quite isolated from their operating environments. This is where the unrealistic assumptions are made about the system's eventual execution model. Such assumptions range from the overtly optimistic (*e.g.*, all executions are instantaneous), to the impractical (*e.g.*, a one-to-one assignment of processes to processors) to the bleakly pessimistic (*e.g.*, all interleavings of process executions are possible). These assumptions rarely hold in practice, and using them to reason about a real-time system's temporal properties can lead to incorrect conclusions.

Also, there has also been considerable progress in developing scheduling algorithms and analyzers [9, 15, 16, 18, 20]. In these approaches, the underlying computational model is generally limited to simple precedence relations between processes where, for the most part, the effect of process synchronization is ignored. Since complex interactions between processes are not captured, these approaches cannot be used for proofs of desirable properties other than schedulability.

We believe it is essential to unify the area of formal methods and scheduling theory. We have made a start in this direction with our development of the CSR specification paradigm [5, 6]. The computation model of CSR is *resource-based*, in that multiple resources execute synchronously, while processes assigned to the same resource are interleaved. Resource contention is resolved by a process' current priority. Using this model, we can prove basic properties of the system design using a proof system, or alternatively, a reachability analyzer.

However, the CSR framework is rather crude, in that possesses a discrete-time model, and only considers the CPU as a resource. Thus it is becomes cumbersome to reason about large-scale, heterogeneous, distributed systems. Certainly more effort is required to investigate the potential links between formal design and schedulability analysis. The high cost associated with real-time failures mandates that various design alternatives can be specified and analyzed *before* implementation.

3 The Problem of Programming Languages

Programming language support is needed to help refine a design into an implementation. Without high-level real-time languages, programmers are frequently forced to use assembly language modules for some of the key components of their systems. Recently, experimental languages have been proposed which provide first-class, real-time constructs [11, 12, 14]. An example of such a construct is "**within** 10ms **do** B," where the block of code "B" must be executed within 10 milliseconds. This constraint is, in turn, conveyed to the real-time scheduler as a directive.

These languages, while providing a convenient framework for *expressing* time in programs, have done little to ease the process of translating a real-time specification into schedulable code. Thus, their timing constructs have not been adopted in any production-level programming languages.

We believe the reason is straightforward: Language constructs such as "**within** 10ms **do** B" establish constraints on *blocks of code*. However, "true" real-time properties establish constraints between the *occurrences of events* [2, 10]. These constraints typically arise from a requirements specification, or from a detailed analysis of the application environment. While language-based constraints are very sensitive to a program's execution time, specification-based constraints must be maintained regardless of the platform's CPU characteristics, memory cycle times, bus arbitration delays, etc.

We have recently taken a new approach to this problem, and our objective is to "bridge the gap" between specification languages and programming languages. Our approach is to treat a real-time program as (1) an event-based, timing specification, which represents the system's real-time requirements; and (2) a functional implementation, that is, the system's code. Instead of constraining *blocks of code*, timing constructs establish constraints between the *observable events* within the code. As an example, consider the following specification fragment, which is rendered pictorially in Figure 1:

(1) The motion-sensor emits **obj_coords** on port **p**.

(2) Transformation function **F** converts **obj_coords** into **next_cmd** for controller.

(3) The controller receives **next_cmd** on port **q**.

(4) To achieve steady state, transmission of **next_cmd** is made no earlier than 3.5 ms after receipt of **obj_coords**.

(5) To guarantee response-time threshold, transmission of **next_cmd** is made no later than 4.0 ms after receipt of **obj_coords**.

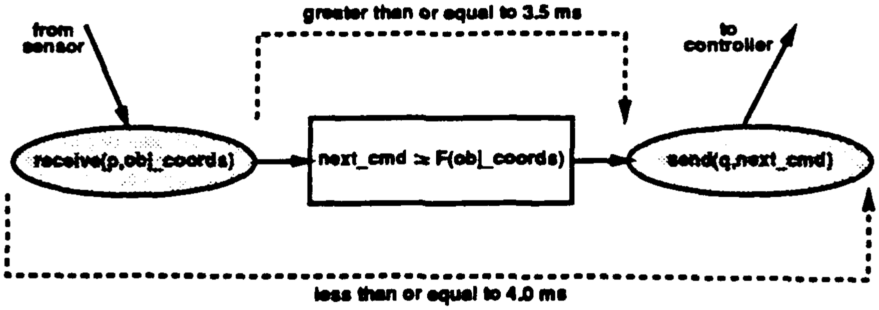

greater than or equal to 3.5 ms
from sensor
to controller
receive(p,obj_coords)
next_cmd := F(obj_coords)
send(q,next_cmd)
less than or equal to 4.0 ms

Figure 1: Event-Based Specification of Sensor-Controller System

We claim that the following program fragments should realize the specification:

```
/* Program A */
do
  receive(p,obj_coords);
start after 3.5 ms finish within 4.0 ms
  {
  next_cmd = F(obj_coords);
  send(q,next_cmd);
  }
```

```
/* Program B */
do
  {
  receive(p,obj_coords);
  next_cmd = F(obj_coords);
  }
start after 3.5 ms finish within 4.0 ms
  send(q,next_cmd);
```

The "**send**" and "**receive**" operations are the system's only observable events. The "**do**" statement establishes timing constraints only between these two operations. On the other hand, the local statement "`next_cmd = F(obj_coords)`" is only constrained by the program's natural control and data dependencies.

Armed with this interpretation, we consider both programs as having equivalent semantics! This is quite different from the approaches mentioned above, where timing constructs establish constraints on code. In that interpretation, program A would first receive its data, then delay for 3.5 ms and finally, evaluate `F` and send the result within the remaining 0.5 ms. Program B would receive its data, evaluate `F`, *then* delay for 3.5 ms and finally, send the result *within 4.0 ms of evaluating* `F` !

Both programs may fail to implement the specification under the code-based constraints. If `F` is a CPU-intensive function (and thereby requires over 0.5 ms of execution time), program A is inherently unschedulable. On the other hand, program B establishes a constraint between the evaluation of `F` and the send operation, and not between the two specified events. Both programs would have to be rewritten to achieve the desired effect. The necessary corrections would include manually decomposing `F`, as well as adjusting the timing constraints. The actual changes would heavily depend on the particular characteristics of the computer, and thus, the very reason for using high-level timing constructs would be defeated.

There are several immediate benefits to this semantics for real-time constructs. First, a source program is not hardware-specific, and thus maintains the abstract, "portable" spirit of a high-level language. Since the timing constraints refer only to specification-based events, they need not be hand-tuned for an individual CPU. Second, this decoupling of timing constraints from code blocks enables a more straightforward implementation of an event-based specification.

Also, much of the arduous, assembly-language level hand-tuning can now be accomplished automatically – by compiler optimization techniques. Many of these are code-motion methods similar to those used in instruction scheduling. Here, however, the objective is to achieve consistency between the real-time constraints and the execution characteristics of the code. In doing this we use the observable events as "signposts," which constrain the places where code may be moved. For example, the local operation "`next_cmd = F(obj_coords)`" can be performed during the delay between the two observable events.

Of course, the greatest challenge lies in the optimization of concurrent programs, since it requires inter-process control-flow analysis. In the end, this problem can be addressed only by close interaction between the compiler and the real-time scheduler. Again, this requires a tight relationship between the development environment and a scheduling tool.

4 Where Should Resources Go?

We see both long and short term goals in the unifying real-time design and implementation. Some very basic technology is required in the short term, such as on-line debuggers and profilers, as well as static timing analyzers. For example, all real-time scheduling theory assumes that real-time response can be reasonably bounded, and schedulability analysis is then carried out using these bounds. Yet no reliable tools exist which can generate these bounds; this is especially true with modern, complex computer architectures.

In the long term, we believe that formal design methods will pay very large dividends. This is particularly true for those efforts aligned with implementation efforts. The "gaps" between design and implementation occur when software engineers get too far removed from system implementers. The most successful projects will span all levels of systems development – through design, integration, testing, operation and maintenance.

References

[1] H. Attiya and N. Lynch. Time Bounds for Real-Time Process Control in the Presence of Timing Uncertainty. In *Proc. IEEE Real-Time Systems Symposium*, pages 268–284,

December 1989.

[2] B. Dasarathy. Timing Constraints of Real-Time Systems: Constructs for Expressing Them, Methods of Validating Them. *IEEE Trans. on Soft. Eng.*, SE-11(1):80–86, January 1985.

[3] J. Davies and S. Schneider. An Introduction to Timed CSP. Technical Report PRG-75, Oxford University Computing Laboratory, Programming Research Group, August 1989.

[4] M.K. Franklin and A. Gabrielian. A Transformational Method for Verifying Safety Properties in Real-Time Systems. In *Proc. IEEE Real-Time Systems Symposium*, pages 112–123, December 1989.

[5] R. Gerber and I. Lee. Communicating Shared Resources: A Model for Distributed Real-Time Systems. In *Proc. 10th IEEE Real-Time Systems Symposium*, 1989.

[6] R. Gerber and I. Lee. A Hierarchical Approach for Automating the Verification of Real-Time Systems. *IEEE Transactions on Software Engineering*, 18(9):768–784, 1992.

[7] R. Gerth and A. Boucher. A Timed Failure Semantics for Extended Communicating Processes. In *Proceedings of ICALP '87, LNCS 267*. Springer Verlag, 1987.

[8] C. Ghezzi, D. Mandrioli, and A. Morzenti. TRIO: A Logic Language for Executable Specifications of Real-Time Systems. *Journal of Systems Software*, 12:107–123, 1990.

[9] K. Hong and J. Leung. Preemptive Scheduling With Release Time and Deadlines. *Real-Time Systems: The Interanational Journal of Time Critical Computing Systems*, 1(3), December 1989.

[10] F. Jahanian and A.K. Mok. Safety analysis of timing properties in real-time systems. *IEEE Transactions on Software Engineering*, SE-12(9):890–904, September 1986.

[11] I. Lee and V. Gehlot. Language Constructs for Distributed Real-Time Programming. In *Proc. IEEE Real-Time Systems Symposium*, 1985.

[12] K. J. Lin and S. Natarajan. Expressing and Maintaining Timing Constraints in FLEX. In *Real-Time Systems Symposium*, December 1988.

[13] N. Lynch and H. Attiya. Using Mappings to Prove Timing Properties. Technical Report MIT/LCS/TM-412b, Laboratory for Computer Science, Massachusetts Institute of Technology, 1988.

[14] V. Nirkhe, S. Tripathi, and A. Agrawala. Language Support for the Maruti Real-Time System. In *Real-Time Systems Symposium*, December 1990.

[15] D. Peng and K.G. Shin. Modeling of Concurrent Task Execution in a Distributed System for Real-time Control. *IEEE Transactions on Computers*, pages 500–516, April 1987.

[16] L. Sha, R. Rajkumar, J. Lehoczky, and K. Ramamritham. Mode change Protocols for Priority Driven Preemptive Scheduling. *Real-Time Systems: The Interanational Journal of Time Critical Computing Systems*, 1(3), December 1989.

[17] A.C. Shaw. Reasoning About Time in Higher-Level Language Software. *IEEE Transactions on Software Engineering*, 15(7):875–889, 1989.

[18] H. Tokuda and M. Kotera. A Real-Time Tool Set for the ARTS Kernel. In *Proc. IEEE Real-Time Systems Symposium*, pages 289–298, December 1988.

[19] V. Yodaiken and K. Ramamritham. Specifying and Verifying a Real-Time Priority Queue With Modal Algebra. In *Proc. 11th IEEE Real-Time Systems Symposium*, 1990.

[20] W. Zhao, K. Ramamritham, and J. Stankovic. Preemptive Scheduling under Time and Resource Constraints. *IEEE Transactions on Computers*, pages 949–960, August 1987.

THE IMPACT OF COMPUTER TECHNOLOGY ON REAL-TIME DESIGN AND REAL-TIME SCHEDULING

Andre Goforth
NASA Ames Research Center
Moffett Field, CA
94035
andy@ptolemy.arc.nasa.gov
415-604-4809

The workshop on large, distributed, parallel architecture, real-time systems has identified four major issues regarding the development of these systems. My position is that a fifth issue underlies these four and must be addressed as part of any solution for the four. This issue is that a breakthrough in standardization and quantification of computer system services critical to this class of real-time systems will have to be made and incorporated into the next generation of methodologies and real-time scheduling theories. The scheduling function and its interface(s) in a distributed, parallel, real-time setting is one example; the system support for expressing abstract notions of timeliness and system response above and beyond the conventional concepts of priority and event is another. Other real-time functions are critical and, no doubt, need to be discussed in more depth in the workshop. The workshop has identified design methodologists and real-time scheduling theorists as the two key players in advancing the state of the practice. Because they treat themselves as separate disciplines these two groups have worked independently for the most part. Each real-time design methodology and real-time scheduling approach has their adherents and critics. Instead of adding to the potential debate between any of these, I submit a corollary to my position: that all current efforts in developing deployable methodologies and scheduling approaches for this class of real-time problem are systemically limited. A third key player needs to be added for design methodology and real-time scheduling theory to have a substantive impact on the industrial base—the computer technologist.

When we, as users and developers of real-time systems, consider the concept of a large, distributed, parallel architecture, real-time system, we are confronted with a situation analogous to that which confronted computer system users and designers of 15 to 20 years ago. At that time, the use of multiple mini-computers was becoming an alternative to the use of centralized mainframe computers. Some of the problems then were how to break the work into meaningful pieces to match the limited performance of a mini-computer and how to orchestrate the processing globally.

There was much debate and question as to whether such systems were a practical alternative to centralized mainframe solutions. For a period of time the only workable alternatives were either to use one vendor's proprietary solution solely or to custom build one's own. Over the long run these alternatives were not effective because of the lack of flexibility and high cost. Today the situation is quite similar—the practical way to design and build (real-time) systems is centralized and to stay close to one vendor's product. However, metaphorically speaking, just because your only tool is a hammer does not mean every problem is a nail.

The obstacle facing pioneers in those days who were looking for flexible and cost effective solutions was the lack of infrastructure—of standardized network protocols, open systems software architectures, and modularized hardware interfaces. Regardless of how elaborate their efforts were in creating design methodologies and computer algorithms for distributed systems, the results were systemically limited. At some point, the development of new standardized (and modularized) interfaces was essential—advanced methodology or algorithm development had reached a point of diminishing returns. For example, in the mid to late 1970s, one key rationale for developing single user workstations as an alternative to time-shared mainframes was to avoid the issues of large, complicated and resource intensive operating systems found on the latter. In hindsight, a good deal of operating systems research and development (along with design methodologies) of that era became as baroque as they did because they were trying to build solutions given near impossible conditions.

We face this situation today. What can be done about this systemic condition? This question is equivalent to the fourth workshop issue which is to identify the most promising technical areas that if funded would improve our capability to design and build such systems. My answer is that there is an opportunity in this workshop for design methodology and real-time scheduling researchers to look hard at the shortcomings in current real-time computing engines that limits the effectiveness of any methodology and any scheduling theoretic approach.

For example, our experiments and performance benchmarks at NASA Ames Research Center corroborate the findings of Norman Howes as discussed in his paper titleα, "Toward a Real-Time Ada Design Methodology". Many methodologies as well as the one explored by Howes are advanced on the basis of desirable design principles. We have found as did Howes that these methods when tried on uniprocessors (and on multiprocessors) can lead to elaborate, highly complex designs that are fundamentally flawed, inefficient or unpredictable in performance. We conclude that little if any attention is given to quantifying the performance of the results of these methodologies. The burden does not all rest on the developers of these methodologies; the computer was a significant accomplice. For example, synchronization primitives failed in special cases; event timer functions performed erratically; in general, performance of real-time functions often was affected by the load and ordering of events. We found this to be the case across three distinct computing platforms/Ada software environments.

Because the technology base for real-time systems is so brittle and quirky there is little incentive for anyone to attempt to develop a methodology that embraces the level of detail necessary to be of substantive use in a real-time project. Any substantive attempt is tantamount to the development of a project-specific tool and, as a consequence, little if any long term leverage is gained. This leads me to the first workshop issue: what methodology would I choose if I had the responsibility of developing a large, distributed, parallel architecture real-time system?

Faced with no alternative of re-scoping the big four requirements—large, distributed, parallel architecture and real-time—I would be justified to expend engineering resources to develop a project specific methodology. The methodology would be based on repetitive use of 'cut and try' that was the staple of design approaches used by electronic engineers in the sixties and seventies. Today, CAD/CAM tools obviate the reliance on 'cut and try' because the physical properties (and performance) are thoroughly understood and hence predictable (for a number of silicon processes and for board level design). We simply do not have a fundamental characterization of real-time computing engine properties today that we can so thoroughly rely upon. As a consequence, the first order of business for the project's methodology is to establish a pedigree of the real-time services of the computing engines that the project can afford. The second is to find real-time scheduling approaches that the computing engines are capable of supporting efficiently while meeting system response requirements. At this point I would expend engineering resources to investigate the augmentation of the project's methodology with desirable design principles. The goal would be to seek a means of unifying or simplifying the overall design or, at least, some intermediate levels of the design.

The 'cut and try' approach is one alternative to the third workshop issue: What is the best method for validating these real-time systems behave as specified? Whether it is the best is a moot point; what the best must have is the element of repetitive use of 'cut and try' where the system, as it is being built up, is incrementally tested against realistic tests cases.

One workshop issue remains: What should be the relationship between real-time Design Theory and real-time Scheduling Theory in a design methodology for this class of systems? To me this is the most problematic issue of all; its resolution is the key to successfully building a new class of real-time systems, and yet our current attempts to address it are systemically limited by the current state of real-time services and interfaces. This point was brought up earlier and is my position for this workshop.

In closing, let us consider what should be the relationship if, for example, there was a real-time computing engine that was capable of supporting a broad variety of real-time scheduling algorithms equally. By this I mean that the differences in observed performance of different algorithms would not be dependent on the implementation but rather on the computational characteristics and complexity of the algorithm. I believe that this would increase substantially the collaboration between Design Theory and Scheduling Theory researchers.

Position Paper on Large, Distributed, Parallel Architecture, Real-Time Systems

Norman R. Howes
Jonathan D. Wood

Computer and Software Engineering Division
Institute for Defense Analyses

1. DESIGN METHODS

During the past three years, we have been considering the problem of what is the best way to design real-time systems, especially for architectures that have more than one processor. We realize that there is probably no one best way. Indeed, there have been a number of proposals by various authors in the recent literature on the subject of real-time systems design. At IDA, we are often in the position of giving advise to our sponsors on methods and techniques to use on real-world projects. Our interest in this area was motivated by our desire to be able to give sound advice in these situations. We originally did not set out to develop a new methodology, but rather to determine, which among the many proposed methods were the most promising.

At the time our investigation started, two methods that were receiving a good deal of attention were the method of Nielsen and Shumate documented in their textbook [9] and the DARTS method of Gomaa [6]. While we have read about many methods, we have only had time to analyses these two in depth in our laboratory. Our experience, based on reading about other experiments (e.g., [7]), and on implementing both Nielsen and Shumate's and Gomaa's examples on sequential and parallel architecture machines and comparing them with implementations of the same examples designed using other methods, leads us to believe that the technique used for process (task) structuring, i.e., what your model of concurrency is based on, is one of the key issues in the design phase.

Both the Nielsen and Shumate method and the DARTS method have been used to develop real-world systems and both methods are evolving (e.g., [10], [11] and [2]). Also, both methods encompass a good deal more than just process structuring. Both of these methods belong to the class of methods that are extensions of the concepts of structured design to the field of concurrent real-time systems. Another class of real-time design methods that is emerging is the class of methodologies based on object modeling (e.g., [1] and [11]). Based on what we have learned so far, it seems that both of these classes of design methods can lead to designs that are unnecessarily complex. The reason for this is probably easier to see in the case of object oriented design techniques. The temptation here is to give each object its own thread of control (process or task). If any of these objects must interact frequently with other objects, it can readily be appreciated that there can be significant overhead in task-to-task communication, synchronizaticn, or both.

So far, all of the methods we have looked at in detail, do not have a good way of insuring that objects that must interact frequently are (usually) in the same thread of control. In fact, many of them have process structuring techniques that seem to encourage placing strongly interacting objects in different threads of control. This not only leads to inefficiency, it leads to designs that are overly complex, and this has ramifications in the area of integration testing, validation and maintenance of the system. It seems to us that what is most needed in these methodologies, is a method for insuring that strongly interacting objects are assigned to the same thread of control whenever possible.

In our lab work, we experimented with trying to base our concurrency model on the real-world processes that were occurring in the problem space for which our system was to operate. The results were improvement of throughput, reduction of number of independent threads of control and reduction in code size, while maintaining or improving timing behavior. Our designs based on this "process modeling" approach also proved to be much more portable than the other examples. All of our experiments were done in the Ada programming language, and we realize that some of the improvement we were seeing can be attributed to compensating for certain inefficiencies with the current version of the Ada language. It is reasonable to believe that in the future, when some of these language inefficiencies have been corrected (e.g., with Ada 9X), that the comparison might not be as dramatic. None-the-less, we believe that the benefits of *process modeling* will still be very noticeable due to the general simplification of design that this method provides with respect to other methodologies we have studied.

2. RELATION BETWEEN DESIGN THEORY AND SCHEDULING THEORY

It is our belief that current real-time scheduling theory is not closely related to most of the real-time design theories that have been advanced recently. Real-time scheduling theory is based on the *time-line model* that models the time frame in which the real-time problem is to be solved. Until recently, the time-line model has been so widely used by real-time practitioners that most of them have *identified* this model with reality. The time-line model is useful for reasoning about how multiple tasks can be scheduled on a single processor and about whether these tasks will meet their deadlines. Because of the importance of predictability in real-time systems, these considerations have been considered so critical, that they have dominated real-time design to the point where real-time design is *conditioned* by schedulability analysis.

It is common practise in real-time design today to divide the time-line up into time slots for the various concurrent tasks comprising the real-time system and assigning *time budgets* to each slot. These time slots are then scheduled using the time budgets for the task execution time using some static scheduling algorithm or by manually fitting the slots into a cyclic executive. The tasks are then designed to try to meet these time budgets. If that proves to be impossible, then an attempt is made to *borrow* additional time from the budgets of other tasks that do not need as large a budget as originally assumed. The problem with the time-line model is that it is not useful for reasoning about some of the fundamental problems associated with the design of real-time systems, such as what functions of the systems should be grouped into a separate threads of control. It is not a good abstraction for reasoning about how these individual processes or tasks should be designed, and it does not generalize well to multiple-processor systems. Furthermore, this approach introduces significant overhead in order to insure predictable timing behavior by forcing the design to comply with an unnatural model.

Instead of just modeling the time frame in which a real-time problem is to be solved (as with the time-line model), it would be useful to be able to model the problem space in which the problem is to be solved. The benefits from object oriented design in the non-real-time problem domain suggest that modeling the problem space yields information that is relevant to how a system should be designed. In the past few years, new design methods for real-time systems have been emerging whose underlying models are either a generalization of the time-line model or a replacement for it. The earliest departure from the time-line model seems to have been in the area of *function-driven* scheduling, e.g., [8] and [13]. Here, a *time value* function or an *importance* function is used to *blur* or *spread* the concept of a dead-line. Next came the class of methodoiogies that are an attempt at generalizing the techniques of struc~~t~~ured analysis and structured design to the real-time domain, e. g., [6], [14] and [9]. Thereafter came the class of methodologies that attempt to extend the concepts of object oriented design to the real-time domain. One recent real-time conference featured over a half dozen papers proposing various "object oriented real-time methodologies". Concurrent with the emergence of the object oriented ral-time methodologies was the emergence of a small number of papers, e.g., [3], [12], [7] and [5] that suggest using a model similar, but somewhat different to the object model that we will refer to as *process modeling*.

We believe all of these generalizations and alternatives to the time-line model supply additional information that may be helpful in the design of real-time systems, but that process modeling offers the greatest potential. Further, it is our position that real-time design theory and real-time scheduling theory should have a closer relationship, and that this should be accomplished by a rethinking of real-time scheduling theory so that eventually it will support these new design methods. At the present time, real-time scheduling theory is usually based on a number of simplifying assumptions that in effect assume that the systems, to which the theory is to apply, have already been designed. For instance, they make the following kinds of assumptions: (1) the (perhaps worst case) execution times of all tasks are known, (2) most of the tasks are periodic, with the occasional need to handle an asynchronous request for service, (3) the deadlines for all tasks are known in advance. As a result, current scheduling theory does not support current design theory very well, and in order to use it, one has to allow the design to be constrained by the time-line model.

More and more, these new design methods are being attempted with the knowledge that the determination that the system will behave predictably and meet its timing requirements will have to be ascertained via testing of the finished design either by prototypes or simulation. While this is a workable approach from a practical point of view, it is somewhat undesirable in that there is no underlying (theoretical or mathematical) reasoning about why the system behaves (with respect to timing behavior) as it does. It is therefore important that scheduling techniques be developed that support design methods rather than force designs by means of unnatural models or be employed as *forcing functions* to force systems designed via other models to try to meet the desired timing requirements.

In the dissertation [13] Strayer essentially argues for real time systems without deadlines that *do the right thing* at each instance of time rather than attempt to meet deadlines. He argues that we can be assured that they do the right thing at each instance of time because they are function driven designs and his importance functions insure that the system is doing the most important thing at any instance of time. While we are not entirely convinced by Strayer's arguments, we believe that the approach he suggests would be superior to trying to meet deadlines, if it can be accomplished via importance functions. One advantage of a system that was designed to do the right thing at any given instant would be that it would therefore always behave in the best possible way during transient overload.

Strayer's concepts are yet to be proven in the real-world of system implementation and testing. However, we believe that an approach along these lines my offer real promise. Several real-time systems developers that we have talked to, never even try to apply current scheduling theory because they believe it to be too restrictive or too unrealistic to merit consideration. Consequently, systems are designed, prototyped, tested, redesigned, retested, etc. until a workable solution emerges. The problem with this approach is that it is not based on well understood (perhaps proven) design principles that are supported by scheduling theory. Then when requirements change somewhat or when the system is to be redesigned to meet similar but different requirements, the whole process has to be started over. One process control company design department we talked to stated that they tried to mitigate this redesign problem by starting with a similar system if possible, modifying it so that it *might* meet the new requirements and then testing it. By starting at this point they were often able to save some time with respect to the alternative of starting from scratch.

3. VALIDATING DISTRIBUTED OR PARALLEL DESIGNS

We do not have much of a position on the validation of such systems, primarily because general experience with validating systems of this class seems to be lacking and our own experience is limited. However, based on our experience measuring the performance of small but representative real-time systems on both single and multiprocessor machines, and redesigning them for better, performance, we believe that the detailed testing of parallel real-time systems offers a wealth of insight into the (often unexpected) behavior of these systems. Consequently, we believe that testing will currently have to play the primary role in verification of systems of this class.

While we are not very knowledgable in the area of formal methods and proving of programs, we think that this area has an important role to play in the future of this class of systems. However, since our experience, and much of the experience of other researchers are with languages that do not have provable semantics, we cannot conceive of how parallel real-time systems could be validated without extensive testing at the present time. On the other hand, attempts at the formal specification of such systems still seems desirable because it provides an unambiguous statement of what the system *should* do which can provide insight into how tests might be constructed to determine if the system behaves as specified or not. Determining how to construct a test to determine if a parallel real-time system meets a specific requirement is often highly non-trivial. Techniques employed for testing of sequential or concurrent systems often do not apply when the system has parallel threads of control. Our experience, for example, with debuggers on parallel machines is that they often cannot trace all the threads of control that are executing simultaneously in a meaningful way. With sequential debuggers the program can be stopped and restarted without affecting the logic of the program. But with a parallel program, what does it mean to stop one of the threads of control while the others continue on? Or what does it mean to stop all threads of control simultaneously, because in practise, this cannot be achieved.

Validation of parallel real-time code might be undertaken with some sort of "non-intrusive" monitor such as the product *parasight* offer by Encore for their parallel machines. We have not been able to experiment with this product because our real-time test beds are written in Ada and parasight currently does not work with the Ada language. Basically, the idea here is that one of the multiple processors is used to gather information in a non-intrusive fashion (that does not affect timing) about the behavior of the code being executed on the other processors.

For our purposes, we have found that code instrumentation works well for answering many of the questions related to behavior of a running system. Our experience seems to indicate that when code instrumentation can be used, it should be designed into the system from the beginning with a view toward always having it there, because its removal alters the timing so there are no guarantees that the non-instrumented code will behave exactly as what was observed in the laboratory with the instrumented code. Our experience indicates that a great deal of information can be learned about the execution behavior of a program with only a small overhead.

Consequently, our position is that at the current state of the practise, designing code instrumentation into the parallel or distributed real-time system specifically for testing if the system meets all or most of its requirements is the best way to do validation. The instrumentation code will remain in the system for the life of the system. If at a later time, the system is changed to meet new requirements, new instrumentation code will have to be added to validate these new features, and the system will have to be revalidated.

4. PROMISING AREAS WHERE RESOURCES MIGHT BE APPLIED

Contrary to popular opinion, we do not believe that it would be profitable to apply resources at this time to support the development of automated design tools for parallel or distributed real-time systems, because we feel that the design process for this class of systems is not yet well enough understood to warrant such tools. Such tools would only help us to make the same mistakes we are currently making at a faster rate. On the other hand, we believe that an investment in automated testing tools that would help us better understand the behavior of executing parallel or distributed systems would be a valuable aid in correcting current design flaws and for learning more about how the behavior of this class of systems is modified by using different design techniques.

Currently, we believe that the tools that would benefit a development project for a system of this class the most are the classical software engineering tools for configuration management and project control which are already readily available. Next, we think that tools that would assist in the simulation and evaluation of design alternatives would provide the most benefit. Such tools for this class of system are not so readily available. Thereafter, would come automated tools for testing and validation which also are not readily available. We believe that both simulation and prototyping are necessary in the project life-cycle for this class of systems at the present time. This iterative approach is time consuming and is where new automated support would provide the most realizable short term gain.

5. REFERENCES

1. A. Agrawala and S. Levi, Real-Time System Design, McGraw-Hill, 1990.

2. M. Cochran and H. Gomaa, *Validating the ADARTS Software Design Method for Real-Time Systems*, Proc. ACM TRI-Ada '91 Conf., San Jose, Oct. 1991, pp. 33 -44.

3. N. Howes and A. Weaver, *Measurements of Ada Overhead in OSI-Style Communication Systems*, IEEE Trans. on Software Eng., Vol. 15, No. 12, pp. 1507 - 1517, Dec. 1989.

4. N. Howes, *Toward a Real-Time Ada Design Methodology*, Proc. ACM TRI-Ada '90 Conf., pp. 189 - 204, Dec. 1990.

5. N. Howes, *Real-Time Ada Design Methodologies and their Impact on Performance*, IDA Paper P-2488, June 1991.

6. H. Gomaa, *A Software Design Method for Real-Time Systems*, Comm. ACM, Vol. 27, No. 9, Sept. 1984.

7. S. Hufnagel and J. Browne, *Performance Properties of Vertically Partitioned Object-Oriented Systems*, IEEE Trans. on Software Eng., Vol. 15, No. 8, August 1989.

8. E. D. Jensen, *The Archons Project: An Overview*, Proc. of the International Symp. on Synchronization, Control and Communication, Academic Press, 1983.

9. K. Nielsen and K. Shumate, *Designing Large Real-Time Systems with Ada*, Multiscience Press, Inc., New York, 1988.

10. K. Nielsen, *Ada in Distributed Real-Time Systems*, McGraw-Hill, New York, 1990.

11. K. Nielsen, *Object-Oriented Design with Ada: Maximizing Reusability for Real-Time Systems*, Bantam Books, New York, 1992.

12. B. Sanden, *Entity-Life Modeling and Structured Analysis in Real-Time Software Design - A Comparison*, Comm. ACM, Vol. 32, No. 12, pp. 1458 - 1466, Dec. 1989.

13. W. T. Strayer, *Function-Driven Scheduling: A General Framework for Expression and Analysis of Scheduling*, Dissertation, University of Virginia, May 1992.

14. P. T. Ward and S. J. Mellor, *Structured Development for Real-Time Systems, Vol. 1: Introduction and Tools*, Yourdon Press, New York, 1985.

A Timeliness Model For Scaleable Realtime Computer Systems

E. Douglas Jensen

Digital Equipment Corporation, Maynard, MA, USA 01754
Voice +1 508 493 1201, Fax +1 508 493 5011, Email jensen@helix.dec.com

Abstract

Many realtime computer system manufacturers and users need products that are *scaleable*: which have consistent interfaces, functional components, and development environments; and which span a wide spectrum—from small, simple, centralized, tactical subsystems to large, complex, decentralized, mission-critical systems. This requires realtime OS's which are highly scaleable in a number of essential respects, whereas all extant ones are only modestly scaleable. A particularly important, and hitherto intractable, form of realtime OS scaleability is the degree of timeliness predictability—i.e., "hardness." The *Benefit Accrual Model* is a framework that generalizes the traditional special cases of deadlines as time constraints, and unanimous optimum as the scheduling criterion; this enables timeliness to be scaled—dynamically—over a wide spectrum of realtime "hardness" and "softness" in a unified way. *Best-effort* scheduling algorithms exploit this generality. The progenitor of this timeliness paradigm was created in 1977 and introduced in the Alpha decentralized realtime OS kernel at Carnegie-Mellon University in 1985; the current version is being developed and incorporated by Digital Equipment into a new version of the Mach 3 kernel for a highly scaleable realtime OS architecture.

1 Introduction

Many realtime computer system manufacturers and users need products that are *scaleable*: which have consistent interfaces, functional components, and development environments; and which span a wide spectrum—from small, simple, centralized tactical subsystems, to large, complex, decentralized, mission-critical systems, as required by any given application.

Suitably high degrees of scaleability benefit the computer manufacturer, solution supplier, and user, by lowering software (and thus system) life cycle costs through such benefits as: wider usability; easier portability and investment protection; and improved adaptability to evolving application needs and technologies—especially valuable in long-life realtime systems.

Realtime software in general, and operating systems in particular, are much more difficult to make scaleable than is hardware. No extant realtime operating system products are more than modestly scaleable, at best; different kinds and degrees of realtime needs are met with different realtime operating systems.

There are many dimensions in which realtime systems and operating systems are more or (usually) less scaleable; especially important ones include functionality, decentralization, performance, predictability (of timeliness), and fault tolerance. Of these, predictability—often informally called "hardness" and "softness"—is the most technically (and sociologically) challenging to make scaleable, because it requires an improved perception and understanding of what "realtime" fundamentally means; an analogy is the improved understanding of gravity that was required to make certain aspects of physics more scaleable.

The conventional realtime dichotomy of "hard" and "soft" realtime is too oversimplified to be scaleable: "hard" as being "deterministic" is an unrealistic special case; and "soft" as being all other cases is imprecise and ad hoc. This paper describes a basis for describing and managing highly scaleable predictability of timeliness, over a wide spectrum of "hardness" and "softness," in a well-defined and unified way: the *Benefit Accrual Model.* To prepare for the description of this model, we first discuss our understanding of realtime, determinism, and predictability. One of the strengths of this model is that it creates the opportunity for employing *best-effort* realtime scheduling algorithms, as well as conventional algorithms; an overview of this topic is provided at the end of this paper.

2 Realtime, Determinism, and Predictability

The traditional realtime viewpoint and terminology arose from the historical emergence of realtime computing in the context of relatively small, simple, centralized, low-level sampled-data subsystems. Realtime systems are popularly dichotomized as "hard" versus "soft." "Hard" realtime conventionally is defined as being "deterministic" in the sense that the only critical computations are those with deadlines, and the scheduling objective is that all these computations must always meet their deadlines, otherwise the system has failed catastrophically. "Soft" realtime conventionally is defined as being "non-deterministic" in the sense that missing a deadline is not necessarily a catastrophic system failure—i.e., "soft" means "not hard:" in some cases, missing certain deadlines under certain conditions may be acceptable; in other cases, the time constraints are not really deadlines but preferred times or time ranges. "Predictability" is commonly regarded as the metric for hardness and softness, although the term is rarely described, much less defined. This traditional realtime viewpoint and terminology is too imprecise, and the resulting resource management concepts and techniques are too oversimplified, to be feasibly scaled up for larger, more decentralized systems.

We consider a computing system or operating system to be a *realtime* one to the extent (this is not a binary attribute) that time—physical or logical, absolute or relative—is part of the system's logic (analogous to errors being states in a fault tolerant system); and in particular, to the extent that resources are managed explicitly to satisfy the completion time constraints of the applications' (and thus its own) computations, whether statically or dynamically.

Time constraints, such as deadlines, are introduced primarily by natural laws—e.g., physical, chemical, biological—which govern an application's behavior and establish acceptable execution completion times for the associated realtime computations. The performance of realtime systems is evaluated in terms of the magnitude of the time constraints which can be satisfied with given computing hardware. Specifically, we define *timeliness* as the metric of how successfully the system is able to satisfy its time constraints.

"Real fast" is often confused with "realtime." A computing system or operating system may satisfy its applications' computation completion time constraints implicitly (by good luck) or by hardware brute force (e.g., MS-DOS on a 200 SPECMARK computer). Such systems may successfully *operate in realtime* and (in the latter case) could be rational, cost-effective solutions for certain applications—but by our definition they are not realtime systems, because they do not employ realtime (time constraint driven) resource management.

Deterministic computation in the realtime context literally means that the computation's timing and timeliness are known absolutely, in advance [1]—there is no uncertainty about any parameters of the computation (e.g., arrival time, execution duration) and its future execution environment (e.g., resource dependencies and conflicts due to other computations) which could affect its timeliness (at least barring faults, and preferably within acceptable fault coverage premises). Thus, deterministic scheduling can—indeed, must [2]—be done off-line. There are very few actual realtime applications and systems which (inherently or forcibly) meet this determinism criterion of absolute timeliness certainty—most are subject to some inevitable dynamic fluctuations and variabilities of computation and communication timing, due to input data arrivals, resource dependencies and conflicts, overloads, and hardware and software exceptions (not to mention faults, errors, and failures outside the presumed coverage).

We regard a computation's timing and timeliness to be non-deterministic but *predictable* in the sense that they can be estimated acceptably; determinism is the maximum, ideal, case [3] which can only be asymmtotically approached in practice (at several kinds of costs). Predictability implies that all parameter values of the computation (e.g., arrival time, execution duration) and its future execution environment (e.g., resource dependencies on, and conflicts with, other computations) are known sufficiently well, and that the computation's timeliness is governed by processes (particularly the scheduler) whose time evolution is sufficiently well controlled. The degree of predictability is then established according to the application-specific interpretation of "acceptably"—e.g., it may be desired that the estimate be extremely precise in most instances at the expense of being less so in the remainder, versus being less but equally precise in every instance.

The timing estimations may be obtained by formal analysis, simulation, empirical measurement, or code examination. The resulting predictability of timeliness (e.g., for response or completion time) may be expressed in a variety of ways—e.g.: an assured upper bound (a lesser or least upper bound since any system's timeliness could be said to be predictable by the choice of one high enough); or in terms of discontinuous rules which relate various execution contexts to estimated, bounded, or even certain timeliness values (those contexts being ones which are most likely, or most important, or just most readily relatable to timeliness estimations); or a probability distribution function of timeliness values.

When the parameters of the computation and its future execution environment are known in the form of random variables so that their uncertainty is characterized by probability distribution functions (a reasonable presumption in many cases), the computation's timeliness may be amenable to stochastic analysis—e.g., the probabilities of execution completion at different times can be derived for certain situations (but as with deterministic scheduling, many of the most interesting cases are either known to be intractable or still defy explicit solution). However, the contexts and thus approaches of stochastic scheduling are predominately oriented toward non-realtime objectives, such as makespan or flowtime [4], which are analytically and computationally easier than stochastic scheduling to meet due times [5] (and for which there is greater application demand than from the realtime community).

The parameters of many realtime systems, especially in higher level, larger scale, and more decentralized contexts, are often too asynchronous—i.e., intermittent, irregular, and interdependent—to have known or tractable probability distribution functions; thus, these systems must be treated as non-stochastically non-deterministic, for which the scheduling technology is still in its infancy.

Independent of the computation and environment parameters, a computation's timeliness predictability also depends on the time evolution characteristics of the scheduler. It is normally taken for granted that realtime scheduling algorithms per se are deterministic even if the parameters are not. Nevertheless, algorithms in general and scheduling algorithms in particular sometimes take advantage (e.g., for simplicity) of making non-deterministic decisions: stochastic schedulers have proven to be successful in certain distributed systems (e.g., [6][7]); and non-stochastic decision making occurs not only in Petri Nets and certain programming languages, but even in realtime scheduling algorithms (e.g., [8]). Most significant, however, is the strong tendency for highly physically [9] and logically [10] decentralized schedulers to enter chaotic regimes [11].

Both determinism and predictability are independent of time constraint (e.g., deadline, response time) magnitudes—a system may be deterministically, or highly predictably, too slow with respect to some particular time constraint magnitude requirement. Thus, timeliness as a realtime performance metric includes both the predictability and magnitude dimensions.

According to our definition of realtime, many computing systems are realtime to some relevant degree: slightly—e.g., a payroll system which automatically generates the checks on time (not early or late); a little more so—e.g., disk driver software; considerably more so—e.g., an OLTP system which automatically performs financial trading based on dynamic market parameters; highly—most (but not all) computing normally thought of as realtime.

Furthermore, the realm of realtime computing is broadening beyond traditional low-level tactical subsystems, to include larger, more complex, more decentralized strategic systems for mission management. This class of realtime application typically coordinates multiple entities which are cooperating adaptively to perform a mission-critical realtime task—such as manufacturing a vehicle, repairing a damaged reactor, conducting an air engagement—despite their individually inaccurate, incomplete views of an inherently dynamic and uncertain application and system state. Under such circumstances, both the application and computing system software (e.g., OS) must make a *best effort* to accommodate dynamic and non-deterministic mission and resource conditions in a robust, adaptable way so as to undertake that as many as possible of the most important computations are as acceptable, in the time and other domains, to the application as possible [9].

There has never been a conceptual or technological framework which could coherently encompass all these degrees of realtime. Consequently, realtime computing concepts and techniques for different systems are ad hoc and largely disjoint from each other, which causes these differences in degree to become differences in kind. This incoherence limits the kinds of realtime systems that can be built, and the cost-effectiveness of those that are built—in particular, it impedes the construction of computing systems which are scaleable in degree of timeliness predictability, and thus in other important dimensions such as functionality, complexity, and decentralization, which require various degrees of predictability.

The developing paradigm of timeliness described here—the *Benefit Accrual Model*—offers a more systematic, general, and realistic framework which we believe can significantly reduce these limitations of classical realtime perspectives and technology. It provides a comprehensive method for expressing time constraints and scheduling objectives that encompasses a wide spectrum of realtime "hardness" and "softness" in a scaleable, unified way.

3 The Benefit Accrual Model Of Timeliness

Introduction

We consider a *realtime computation* to be a segment of a computational entity (such a thread, task, or process) subject to a completion time constraint (such as a deadline).

We define a *time constraint* to be: the specification of: a time period during which completion of the realtime computation's execution affects the temporal component of its acceptability; and that affect (e.g., completing before the deadline is acceptable, and otherwise is unacceptable).

A time constraint is manifest in the computation program as a demarcated region of code whose execution completion time is subject to the time constraint. A computational entity may include multiple realtime computations—sequentially or concurrently (i.e., nested), as shown in Figure 1.

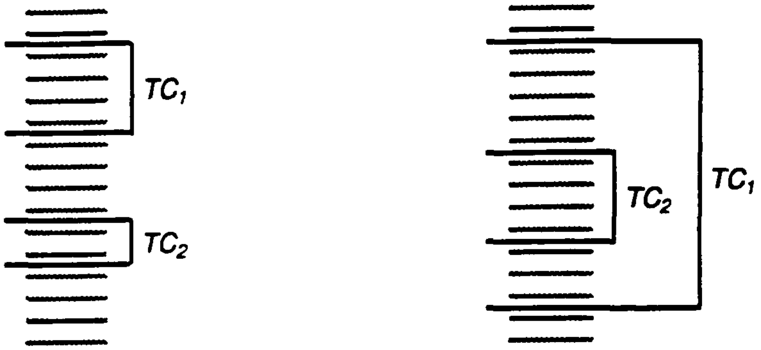


Figure 1: *Time constraints manifest as demarcated code regions*

The classical deadline imposes a binary partitioning of a computation's completion time range into either acceptable (prior to the deadline), or not (after the deadline), as illustrated in Figure 2. The semantics of "not acceptable" are specific to the computation and application—e.g., non-productive or counter-productive in some way.

Often non-deterministic execution-time variabilities make it very useful to have "softer"—in the sense of non-binary—relationships between when a realtime computation completes execution, and the temporal acceptability of that computation. A realistic example of such a softer time constraint is that if a particular computation cannot be completed at an optimum time—i.e., before its "deadline"—then: completing it a little tardy is suboptimum, but better than not completing it at all; however, completing it very tardy is worse than not completing it at all. See Figure 3.

The description of this example indicated that the "deadline" was redefined to be the end of the optimum, rather than the acceptable, completion time zone. The normal definition of deadline (Figure 4) would cause popular realtime scheduling algorithms to complete more computations in the suboptimum zone than was intended by the example soft time constraint. Thus, such non-binary completion time/acceptability relationsbips raise questions such as: which time is best considered the "deadline," and what the other completion delimiting times are; how are these specified times used for scheduling.

The execution of each realtime computation is not necessarily scheduled to maximize its individual temporal acceptability. A realtime system (normally) has a multiplicity of realtime computations which are executed in a partial order according to a *scheduling criterion*: a collective temporal acceptability criterion for a set of realtime computations, in terms of their individual time constraints—e.g., the classical "hard realtime" criterion that all realtime computations meet all their deadlines. In some cases—such as the classical "hard realtime" one—there is an equivalence between the individual and collective temporal acceptability criteria (e.g., "each" and "all").

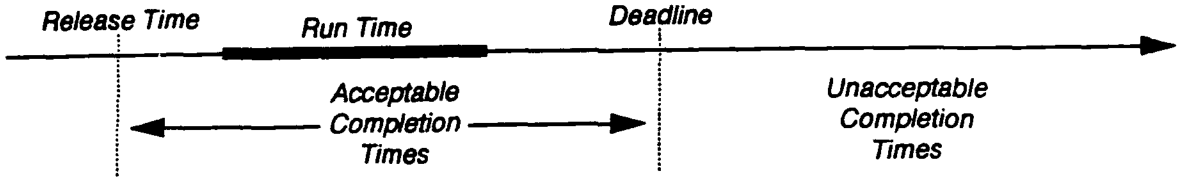


Figure 2: *The classical deadline*

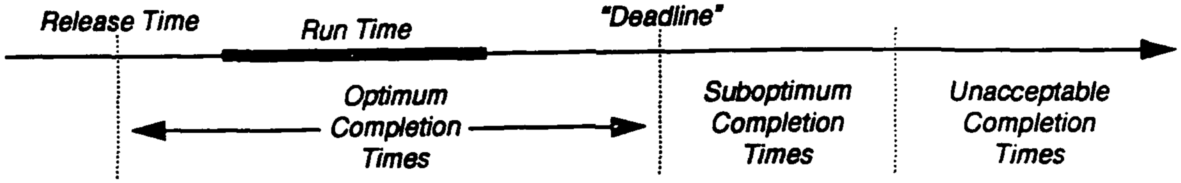


Figure 3: *A "softer" time constraint*

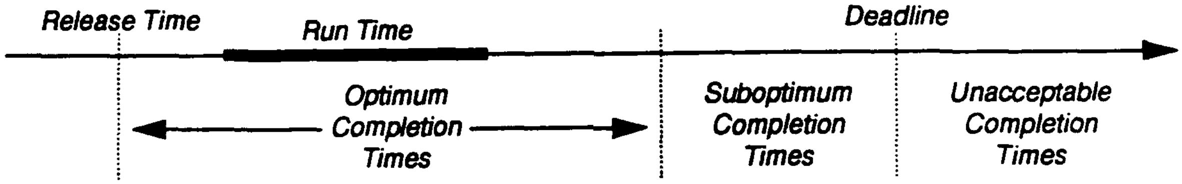


Figure 4: *Combination deadline and softer time constraint*

A particular scheduling criterion applied to a particular set of realtime computations may result in a subset of them whose individual time constraints will/would not be optimally satisfied; how and when this is resolved is situation-specific (the classical "hard realtime" criterion usually implies this condition is an overload which must be avoided á priori).

The traditional "hard realtime" scheduling criterion is a single special case which does not apply to non-binary time constraints, such as those which have multiple completion time zones or redefined "deadline." "Softer" time constraints—in the sense of non-binary completion time acceptability—necessitate associated "softer"—in the sense of non-unanimous and non-optimum—scheduling criteria. In the context of our example, the softer criterion is that the maximum possible number of computations complete in the optimum zone, and all the remainder complete in the suboptimum zone.

Traditional "soft" realtime scheduling criteria are disparate, ad hoc, and imprecise; thus, they do not offer a basis for systematically expressing scheduling criteria for non-binary time constraints.

In the Benefit Accrual Model, a time constraint is a generalization of the conventional "hard deadline" because the conventional "deadline" and "hard realtime" scheduling criterion involve time directly and are well-defined (contrary to the state of conventional "soft" realtime).

The Benefit Accrual Model is based on two concepts: a ***benefit function***, which generalizes the classical "deadline" of a realtime computation; and a ***benefit accrual function***, which generalizes the classical "hard realtime" scheduling criterion that a set of computations always meet all its deadlines.

This model generalizes the author's earlier concept of "time-value function" resource scheduling [12][13], which was first employed in the Alpha realtime decentralized OS kernel [14][15].

Benefit Functions

The *urgency*—i.e., time criticality—of a realtime computation is expressed in terms of the benefit it provides to the system as a function f_B of the time at which the computation is completed (see Figure 5). The benefit metric is application-specific and defined system-wide. Benefit functions are derived by the programmers directly from the requirements and behavior of the realtime computation (usually an application activity); this is subject to a system-wide engineering process (just as are assignments of classical priorities).

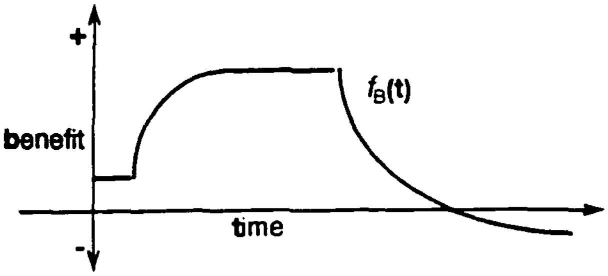


Figure 5: *Benefit function*

The function f_B is *unimodal* if it is concave downward (we will define that linear functions are so)—i.e., any decrease in value cannot be followed by an increase—otherwise it is *multimodal*. A multimodal function has at least one instance of a monotonic decrease in value followed by a monotonic increase, and thus there are multiple non-contiguous *time intervals* when it is better to complete the computation than during the times separating them (see Figure 6). A multimodal function involves non-linear optimization which is often intractable on-line, so we do not discuss multimodal functions further here.

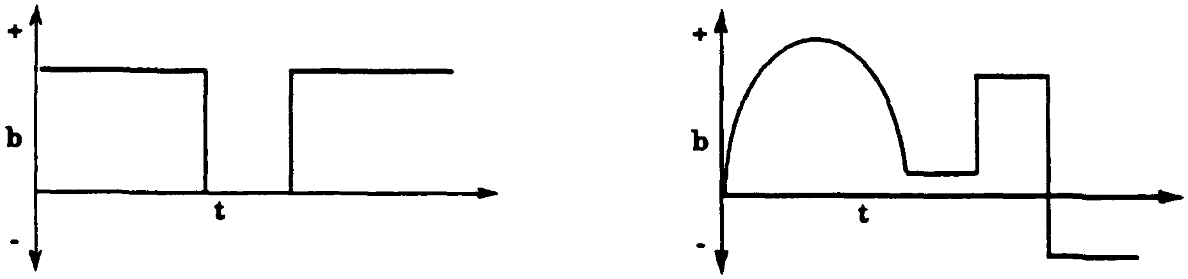


Figure 6: *Multimodal benefit functions*

A computation's benefit function can be changed each time it is released for execution, as illustrated in Figure 7.

The benefit function time axis is the one the scheduler uses. It may be physical, either absolute ("calendar/wall clock") time—i.e., year, month, date, hour, minute, second, mSec, µSec—or relative to (since) some past event. Alternatively, it may be logical—e.g., a number which monotonically increases, but not necessarily at regular intervals. In some distributed realtime computer systems, time constraints can span nodes, which requires a trans-node time frame (global clock). The origin of the benefit function axes is the *current time* (value of the system clock) t_C, as seen in Figure 8.

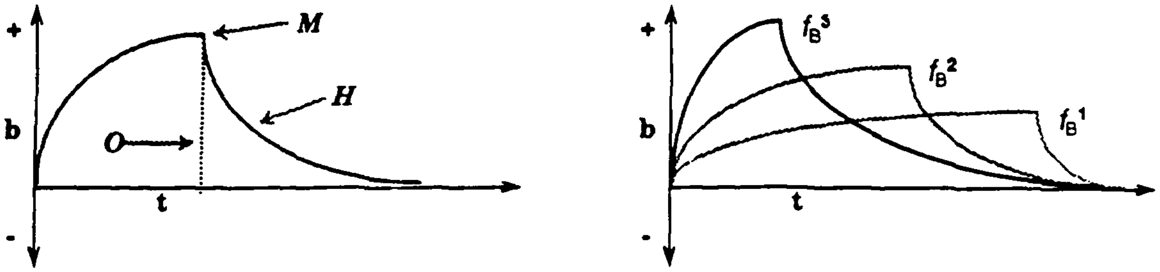


Figure 7: *A computation changing benefit functions each release*

It may be preferable for an application programmer to express some benefit functions in terms of a time parameter different from that of the global time axis—e.g.: a computation's deadline being incremental time units from now, regardless of the axis metric; or a particular physical absolute time, though the axis is physical relative time—but these differences must subsequently be translated for scheduling. Translations between physical and logical time frames are ordinarily infeasible.

Expressing a benefit function relative to a future time/ event, such as the completion of some other computation, or an external signal, is adding a (generally dynamic) dependency to the time constraint. Dependencies must be accommodated in conjunction with time constraints according to some specific scheduling policy, and thus are not part of the Benefit Accrual Model per se.

The earliest time for which a benefit function is defined is called its *initial time* t_I;the latest time for which a benefit function is defined is called its *terminal time* t_T (see Figure 8). Some systems and scheduling algorithms call for the specification of an indefinitely extended terminal time. A benefit function is evaluated only for values of its time parameter between the current time and its terminal time. If the terminal time is reached ($t_T = t_C$) and execution of the realtime computation has not begun or has begun but not completed, the realtime computation is aborted and the time constraint is removed from scheduling consideration. If a realtime computation is sufficiently likely to complete execution after its initial time, a scheduling algorithm could choose to begin it before the initial time.

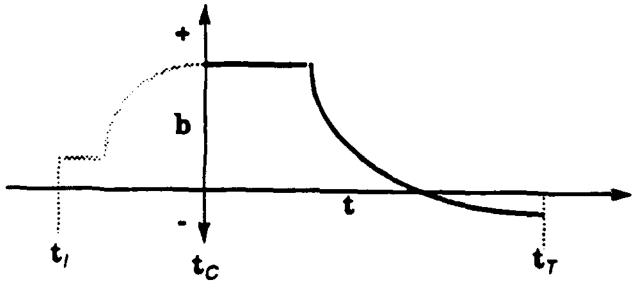


Figure 8: *Initial and terminal times*

The *later time* t_L (see Figure 9) is that after which the benefit function value is (monotonically) non-increasing; thus, completing the realtime computation at or after this time is better. A benefit function always has a later time. The *sooner time* t_S is that after which the benefit function value is (monotonically) decreasing; thus, completing the realtime computation at or before this time is better. A benefit function need not have a $t_S < t_T$. If its value becomes zero or negative at time $t_E \geq t_S$, a benefit function has an *expiration time*.

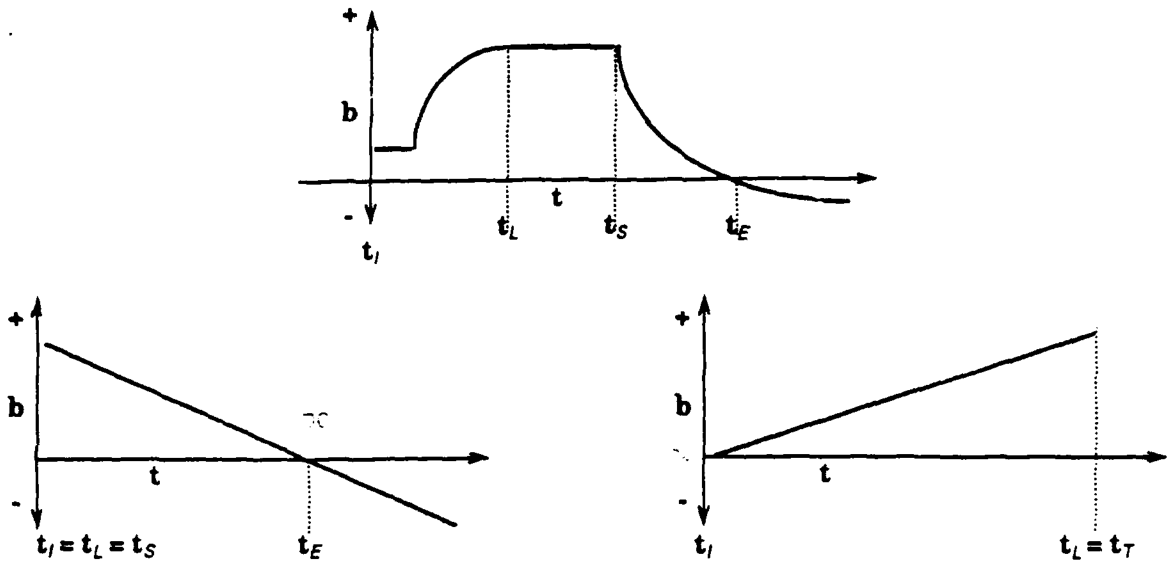


Figure 9: *Later, sooner, and expiration times*

It can be necessary for a realtime computation to be completed at a time yielding zero or negative benefit: *early*, rather than delaying execution until the greatest positive benefit is expected; or *tardy*, rather than terminating (or not initiating) execution after there is no expectation of positive benefit. Such cases arise due to dynamic dependencies, when a computation: has been initiated and cannot be stopped (preempted or aborted) or undone (such as one related to a physical activity in the application environment); or would block another if not completed, despite its consequential zero or negative benefit.

A special case of a sooner time t_S is a *due time* t_D, distinguished by the benefit function's first derivative having an infinite discontinuity at $t_S = t_E$ (shown in Figure 10). A *deadline* is a due time subject to a collective temporal acceptability criterion which does not allow the due time to be missed.

A benefit function is defined as *hard* if it has: a zero or constant negative value before t_L; an infinite discontinuity in its first derivative at t_L if $t_L > t_I$; a due time t_D; a constant value between t_L and t_D; and a constant value between t_D and t_T.

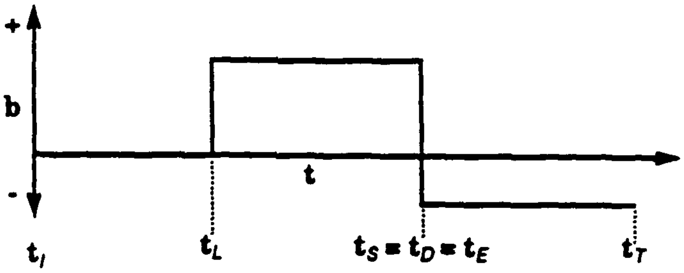


Figure 10: *Example hard benefit function*

The most common meaning of a classical "hard deadline"—a computation which completes anytime between its initial and deadline times is uniformly acceptable, and otherwise is unacceptably tardy—corresponds in this model to a hard benefit function with deadline $t_D = t_T$ and unit binary range {0,1} (Figure 11). Classical definitions of "hard deadline" vary a little: they generally do not provide for a $t_L > t_I$; sometimes the range of this function is $\{-\infty, 1\}$; a few algorithms define the range as $\{0, ke\}$, where e is the computation's execution duration and k is a proportionality factor; many systems allow phases within each period to be arbitrary, while others require all the phases to be synchronized; most deterministic algorithms, such as rate-monotonic, require the highest priority ready activity to execute, thus disallowing phase shifts.

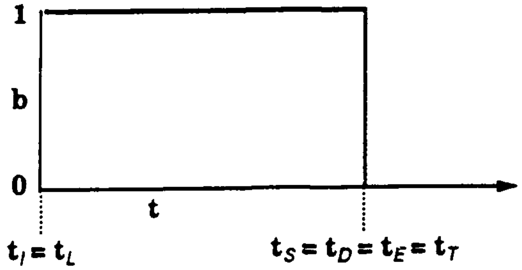


Figure 11: *Hard deadline benefit function*

All benefit functions which are not hard are *soft*. Soft benefit functions can have arbitrary values before and after the optimal value at t_S (Figure 12);. they need not have constant values on each side of t_L and t_D, or expiration times (Figure 13).

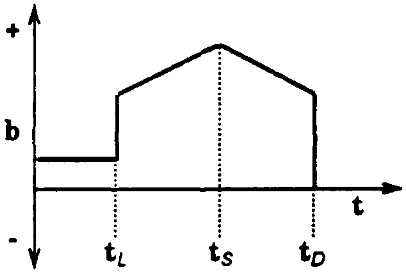


Figure 12: *Example soft benefit function*

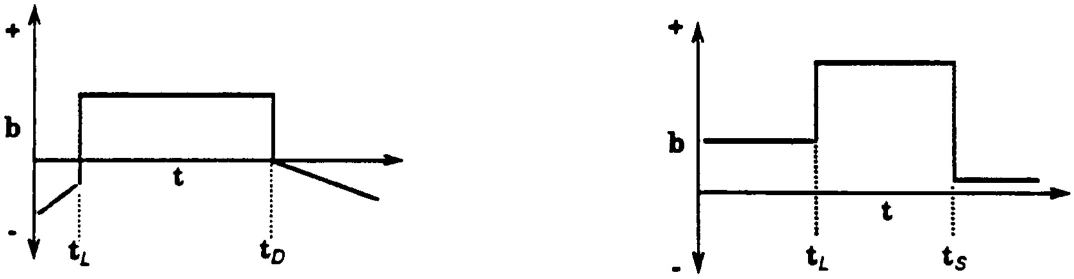


Figure 13: *Example soft benefit functions*

A time constraint—and thus benefit function—is made known to the scheduler at its *release* time (which is usually a scheduling event).

When the benefit function is released, its initial time may be either the current time—the time constraint is released at the time it is to take effect (i.e., at $t_I = t_C$)—or a future time—the time constraint is released in advance (i.e., $t_I > t_C$) to improve scheduling (but $t_I \leq t_C$ is a necessary condition for the computation to complete, if not also begin, execution). See Figure 14.

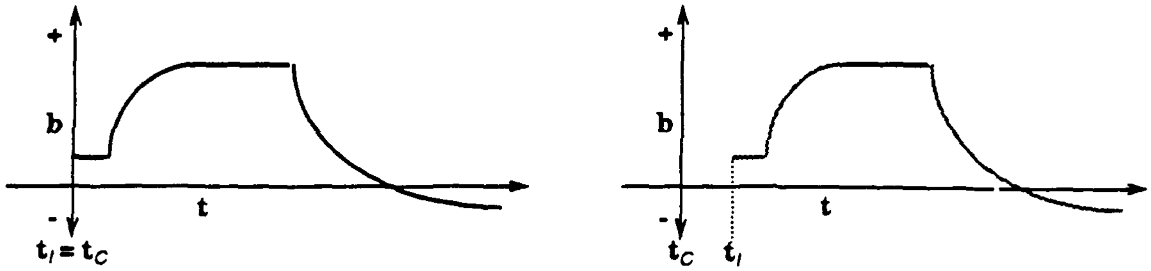


Figure 14: *Tne initial; time may be either the current time or a future time*

Expressing or releasing a benefit function relative to a future time/event, such as the completion of some other computation or an external signal, is adding a (generally dynamic) dependency to the time constraint. Dynamic dependencies can require a realtime computation to be completed at a time yielding zero or negative benefit—for example, when a computation: has been initiated and cannot be stopped (preempted or aborted) or undone (such as one related to a physical activity in the application environment); or would block another if not completed, despite its consequential zero or negative benefit. Dynamic dependencies can require indefinitely extended function terminal times. Dependencies must be accommodated in conjunction with time constraints according to some specific scheduling policy, and thus are not part of the benefit accrual model per se.

Importance

Each computation generally also has a relative *importance*—i.e., functional criticality—with respect to other computations contending for completion. Importance is orthogonal to urgency: a computation with high urgency (e.g., a near deadline) may not be highly important; or a computation with low urgency (e.g., a far deadline) may be very important.

Importance may be a function f_I of time and other parameters that reflect the application and computing system state, and can be represented and employed similar to urgency (Figure 15).

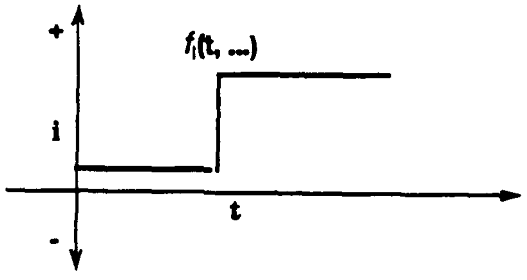


Figure 15: *Importance function*

In simple cases, importance may be a constant, and urgency (benefit) may be simply scaled by importance—e.g., by multiplication, addition, or concatenation. In more general cases where importance needs to be a variable, f_B and f_I must be evaluated together dynamically to determine the benefit—e.g., as some function of the f_B and f_I functions, $g(f_B, f_I)$. See Figure 16.

Execution Duration

A realtime computation has an execution duration e which the scheduler usually has some information about prior to execution. This information can be either known deterministically (the most common presumption), or estimated. Most estimates are stochastic (known in expectation), but alternatively may be non-stochastic—e.g., bounds or rules. Execution duration in-

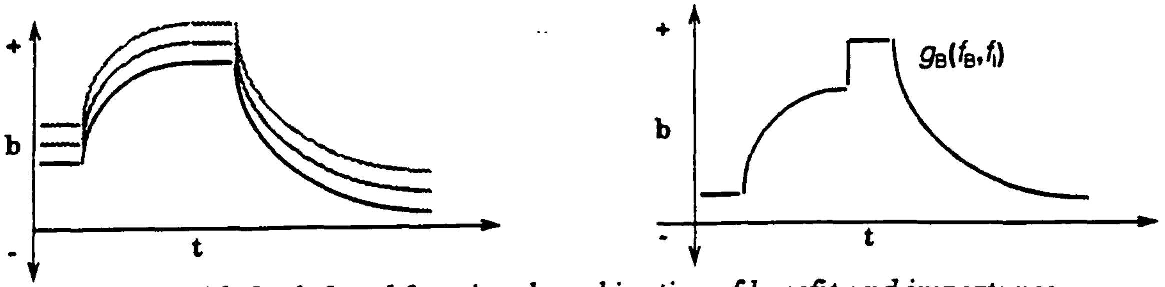


Figure 16: *Scaled and functional combination of benefit and importance*

formation may or may not take into account a forecast of dynamic dependencies. Non-deterministic durations may be estimated dynamically (during the computation's execution)—e.g., conditional probability distributions, or execution-time knowledge-driven rules.

Benefit Accrual Functions

The scheduler considers all released time constraints between the current time and its *horizon* t_H—the future-most terminal time (Figure 17). It assigns the estimated execution completion times, and consequently the initiation times and ordering, for those computations using an algorithm which seeks to sufficiently satisfy the scheduling (collective temporal acceptability) criterion (such as earliest-deadline-first for the classical "hard realtime" criterion of all computations meeting their deadlines). The algorithm should also take into account dependencies and importances.

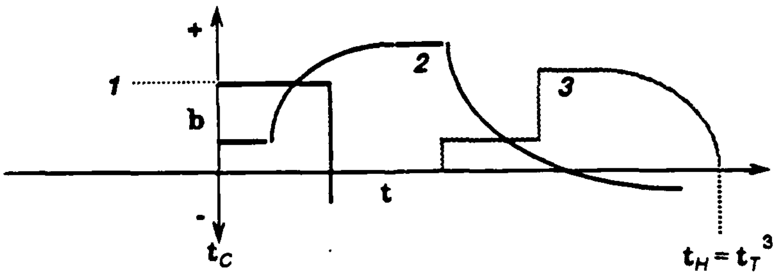


Figure 17: *The scheduler considers all released benefit functions to its horizon*

It is *feasible* to schedule a particular set of realtime computations if its collective temporal acceptability criterion can be sufficiently satisfied. A particular set of realtime computations is *schedulable* if there exists at least one algorithm which can feasibly schedule it. A scheduling algorithm is *optimal* if it always produces a feasible schedule whenever: in the static case, any other algorithm can do so; in the dynamic case, a static algorithm with complete á priori knowledge would do so.

The ideal case of every computation always completing execution at an optimum time is unrealistic in general. Even though the traditional "hard realtime" cases are intended—and commonly imagined—to achieve this ideal, physical laws (especially in asynchronous decentralized systems) or the intrinsic nature of the applications (especially at mission management levels) generally make it non-cost-effective or even impossible.

Most actual realtime systems desire a sufficient number of computation completion times to be sufficiently likely to be sufficiently acceptable (perhaps optimal) under the current application and computer system circumstances.

For the special case of any collective temporal acceptability criterion defined to be a unanimous optimum of the individual temporal acceptabilities, there is an equivalent criterion defined in terms of individual, rather than collective, optimums—e.g., meet all deadlines means meet each deadline, and maximize all benefits means maximize each benefit.

In general, collective temporal acceptability is not defined as necessarily unanimous or optimum with respect to the individual computations' temporal acceptability—e.g., minimize the number of missed deadlines, or maximize the sum of the benefits.

In the Benefit Accrual Model, collective temporal acceptability criteria are based on accruing benefit from the individual computations in a set, in a manner specified by a benefit accrual function for that set. This is general enough to encompass a wide range of temporal acceptability criteria—e.g.: the optimum cases such as traditional "hard realtime," for which the function is the product of the individual benefits (assuming the usual range of {0, 1}); potentially suboptimum cases, for which example functions are to maximize the sum (mean, etc.) of the individual benefits; the number of computations during a time frame T which achieve at least P percent of their maximum possible benefit; the probability that at least P percent of the computations during a time frame T will achieve their maximum benefits. Collective temporal acceptability criteria can be employed for scheduling or performance evaluation.

4 Best-Effort Scheduling

Introduction

Scheduling principles and practices which are realtime by our definition (i.e., based on satisfying completion time constraints) have until recently been focused exclusively on guaranteeing that a unanimous optimum scheduling criterion will be met (e.g., the classical "hard realtime" case of guaranteeing that all deadlines are always met). Even though the traditional "hard realtime" cases are intended—and commonly imagined—to achieve this ideal, physical laws (especially in decentralized systems) or the intrinsic nature of the applications (especially at mission management levels) generally make it either non-cost-effective or impossible (there are only a few exceptions).

In general, realtime systems need a sufficient number of computation completion times to be sufficiently likely to be sufficiently acceptable (perhaps optimal), given the current application and computer system circumstances (perhaps over a wide range of such circumstances)—where each instance of "sufficient" is application-specific.

The Benefit Accrual Model provides a framework for expressing "softer" time constraints—in the sense of non-binary completion time acceptability—and scheduling criteria—in the sense of non-unanimous and non-optimum. It accomplishes this in addition to—and in the same manner as—the conventional "hard" time constraints and scheduling criteria. These softer needs are realized with *best-effort* scheduling algorithms

Best-effort (BE) realtime scheduling algorithms aggressively seek to provide the "best"—as specified by the application— computational timeliness they can, given the current application and computer resource conditions. Best-effort resource management is generally heuristic—a familiar approach at the application levels (most conspicuously in artificial intelligence, pattern recognition) and less visibly at the system software levels. Because heuristics are essentially foreign in traditional realtime systems, we employ the term "best effort" to more clearly evoke our intended departure in philosophy—analogous to the utilization of the term "guess" [16] for inferences performed by certain intelligent user interfaces, e.g., [17].

Heuristics in general, and best effort realtime resource management in particular, involve trade-offs of risk management and situational coverage. Best-effort on-line realtime scheduling heuristics currently offer empirically-based high confidence that acceptable computational timeliness will be achieved over a broad range of conditions; but with no or low formal bounds on guaranteed timeliness (note that this is necessarily always true of the humans currently performing best-effort resource management). Conversely, traditional "hard" realtime scheduling algorithms provide formal guarantees of optimal computational timeliness under extremely restricted—generally unrealistic—conditions, but behavior which is unknown or known to be pathologically wrong outside those conditions. Examples of applications which seem to call naturally for each of these extremes come immediately to mind—but in making the trade-offs and compromises to find an application-specific appropriate middle ground, one must beware of the human trait to undervalue the reduction, as opposed to the elimination, of risks [18].

Overview of Best-Effort Realtime Scheduling Work

This concept, and the Time-Value Function progenitor of the Benefit Accrual Model as a framework for expressing time constraints, were originated by Jensen [12][13]. The first generation of BE—on-line (at execution time)—scheduling algorithms emerged from Jensen's Archons Project at CMU [19], for the Alpha realtime decentralized OS kernel [14]: Locke's algorithm [7] and Clark's algorithm [8].

Locke's algorithm allows a wide variety—but not all forms—of Time-Value Functions (TVF's). Locke intends that importance be reflected by scaling the TVF values. The scheduling optimality criterion is the special (but reasonable) case of maximizing the sum of the job values attained. Execution times are defined stochastically.

The algorithm schedules jobs Earliest-Deadline-First (EDF) since that is optimal when underloaded. If a job arrival, or execution time overrun, results in a sufficiently high probability of overload, jobs are set aside in order of minimum *expected value density* (expected value/expected remaining execution time) until the probable overload is removed.

Locke's algorithm does not address dependencies (e.g., precedence, resource conflicts).

Locke used simulations to demonstrate that his algorithm performed well in comparison to others for a number of interesting overload cases, but provided no formal performance characterizations. Versions of Locke's algorithm have been implemented and experimentally verified to be superior and cost-effective with respect to traditional realtime scheduling algorithms for a number of interesting cases. In the Alpha realtime distributed OS kernel, these included: a battle management application for air defense, by General Dynamics and the Archons Project at CMU [20]; and a ball-and-paddle realtime scheduling evaluation testbed by the Archons Project [21]; the Alpha version also added nested time constraints and timeliness failure abort processing. Locke's algorithm was implemented In the Mach 2.5 OS kernel, and measured on a synthesized realtime workload by the Archons Project [22].

Clark's algorithm makes a major contribution by dealing with dependencies (e.g., precedence, resource conflicts) which are not known in advance. It employs the same scheduling optimality criterion as Locke's. Clark permits only rectangular TVF's, whose value is the job's importance. Job execution times are both fixed and known.

Clark's algorithm selects jobs to be scheduled in decreasing order of value density (VD), and then selected jobs are scheduled EDF—for the TVF's he allows, this both meets all deadlines and maximizes summed value. When each job is scheduled, so are those on which it depends. If necessary, precedent jobs are aborted or their deadlines are shortened (whichever is faster), to satisfy the deadline of the dependent job.

Clark used formal analysis and simulations to show that when overloaded, if the algorithm can apply all available cycles to jobs that complete, no other algorithm can accrue greater value given the current knowledge; but since future jobs are unknown, there is no performance guarantee. Clark's algorithm is being implemented in both the Alpha and Mach 3 OS kernels.

A second generation of on-line BE algorithms is being devised as part of a recent multi-university effort to establish formal performance bounds for on-line algorithms in general and certain BE ones in particular [23][24][25]. Their work is focused on the *competitive factor*, which measures the value an algorithm guarantees it will achieve compared to a clairvoyant scheduler.

Like Clark's, their algorithms allow only rectangular TVF's, and (mostly) require both fixed and known execution times.

The principal result is that if all values are proportional to execution time, an on-line algorithm can guarantee a competitive factor of no more than 1/4. The performance bound is lower when value is not proportional to execution time, or the ratio of maximum to minimum VD increases, or execution times are not fixed and known.

This confirms the intuition that realtime performance guarantees are impossible if workload characteristics are unknown. However, the most recent research suggests that acceptable performance assurances may be possible when limited, reasonable, workload information is known; learning and understanding such trade-offs is one of the most important advances still to be made for BE algorithms.

Maynard's thesis [26] is improving the understanding of the overload behavior of on-line realtime scheduling algorithms, and developing techniques for defining benefit functions to yield desired overload behavior. Its scope includes BE schedulers that use benefit density as the load shedding criterion. The work to date provides an algorithm for setting job importance values to impose a strict priority ordering among selected groups of jobs. This allows integration of results from off-line schedulability analysis, to both provide "guarantees" when necessary and possible, and retain the adaptability of dynamic scheduling. His simulations support the validity of the approach. He is also creating tools which help the system designer select and adapt suitable scheduling algorithms for specific applications, and choose appropriate job importance values.

The most closely related work to BE realtime scheduling is Cost-Based Scheduling for queueing and dropping network packets [27]. In this context, a cost function specifies the cost per unit length of queuing delay for a packet as a function of time. Packets are limited to non-decreasing cost functions.

Unlike BE processor scheduling, which create a whole schedule, the cost-based network algorithm queues the next packet which it estimates would cost the most to delay. Cost is calculated using a estimation of future cost that would be incurred, which is the same for all packets. The optimization objective is to minimize the average delay cost incurred by all packets. Dependencies are not considered, but explicitly recognized as critical.

Their simulations show that the algorithm performs well compared to the standard packet queuing algorithms, and to Locke's algorithm, for packets averaging unit length, in near fully loaded conditions. The premises of this work do not correspond well to the workload characteristics of interest for best-effort realtime computation scheduling.

In addition to this on-line research, a first generation of off-line BE algorithms is being devised in France [28][29].

Benefit functions employ more application-supplied information, and thus exact a higher computational price than when little such information is used (e.g., by static priority) or no information is used (e.g., by round robin). Best-effort realtime scheduling algorithms utilize more application-supplied information than is usual, and place specific requirements on the kind of scheduling mechanisms that must be provided (i.e., in the OS kernel—cf. those of the Alpha kernel).

These prices can be minimized by good engineering, and then paid in different ways, including with inexpensive hardware: higher performance processors; a dynamically assigned processor in a multiprocessor node; or a special-purpose hardware accelerator (analogous to a floating-point co-processor) in a uniprocessor or multiprocessor node.

5 Acknowledgments

This work was funded in part by the USAF Rome Laboratories and the USN Surface Warfare Center, as well as by Digital Equipment Corporation.

The author gratefully acknowledges his helpful discussions with Ray Clark, Mike Davis, Alan Downing, Ira Goldstein, David Maynard, and John Peha.

6 References

[1] Lawler, E.L., J.K. Lenstra, and A.H.G. Rinnoy Kan, *Recent Developments in Deterministic Sequencing and Scheduling: A Survey*, Deterministic and Stochastic Scheduling, M.A.H. Dempster et al. (eds), D. Reidel, 1982.

[2] Xu, J. and D.L. Parnas, *On Satisfying Timing Constraints in Hard-Real-Time Systems*, Proc. ACM SIGSOFT '91 Conference on Software for Critical Systems (and ACM Software Engineering Notes), December 1991.

[3] Pinedo, M., *On the Computational Complexity of Stochastic Scheduling Problems*, Deterministic and Stochastic Scheduling, M.A.H. Dempster et al. (eds), D. Reidel, 1982.

[4] Weiss, Gideon, *Multiserver Stochastic Scheduling*, Deterministic and Stochastic Scheduling, M.A.H. Dempster et al. (eds.), D. Reidel, 1982.

[5] Gifford, T., *Algorithms for Stochastic Scheduling Problems with Due Dates*,

[6] Glorioso, R.M. and F.C.C.Orsorio, *Stochastic Automata Models in Computer and Communication Networks*, Ch. 9 in Engineering Intelligent Systems, Digital Press, 1980.

[7] C.D. Locke, Best-Effort Decision Making for Real-Time Scheduling, Ph.D. Thesis, CMU-CS-86-134, Department of Computer Science, Carnegie Mellon University, 1986.

[8] R.K. Clark, Scheduling Dependent Real-Time Activities, Ph.D. Thesis, CMU-CS-90-155, School of Computer Science, Carnegie Mellon University, 1990.

[9] Jensen, E.D., *Asynchronous Decentralized Computer Systems*, Proceedings of the NATO Advanced Study Institute on Realtime, Springer-Verlag, 1993.

[10] Jensen, E.D., *Decentralized Control*, Distributed Systems: An Advanced Course, Springer-Verlag, 1981.

[11] Hogg, T. and B.A. Huberman, *Controlling Chaos in Distributed Systems*, Transactions on Systems, Man, and Cybernetics, IEEE, November/December 1991.

[12] Stewart, B., Distributed Data Processing Technology, Interim Report, Honeywell Systems and Research Center, March 1977.

[13] Jensen, E.D., C.D. Locke, and H. Tokuda, *A Time-Value Driven Scheduling Model for Real-*

Time Operating Systems, Proceedings of the Symposium on Real-Time Systems, IEEE, November 1985.

[14] Northcutt, J. D., Mechanisms for Reliable Distributed Real-Time Operating Systems—The Alpha Kernel, Academic Press, 1987.

[15] Clark, R.K., E.D. Jensen, and F.D. Reynolds, *An Architectural Overview of the Alpha Real-Time Distributed Kernel*, Proc. of the USENIX Workshop on Microkernels and Other Kernel Architectures, April 1992.

[16] Myers, B.A., *Demonstrational Interfaces: Step Beyond Direct Manipulation*, Computer, IEEE, August 1992.

[17] Huff, K. and V. Lesser, Knowledge-Based Command Understanding: An Example for the Software Development Environment, TR 82-6, Computer and Information Sciences, U. of MA, 1982.

[18] D. Kahneman, P. Slovic, and A. Tversky (Ed.), Judgement Under Uncertainty: Heuristics and Biases, Cambridge University Press, 1982.

[19] Jensen, E.D., *The Archons Project: An Overview*, Proceedings of the International Symposium on Synchronization, Control, and Communication, Academic Press, 1983.

[20] Maynard, D.P., S.E. Shipman, R.K. Clark, J.D. Northcutt, R.B. Kegley, B.A. Zimmerman, and P.J. Keleher, An Example Real-Time Command, Control, and Battle Management Application for Alpha, Technical Report TR 88121, Archons Project, Computer Science Department, Carnegie-Mellon University, December 1988.

[21] Northcutt, J.D., R.K. Clark, D.P. Maynard, and J.E. Trull, Decentralized Real-Time Scheduling, Final Technical Report, Contract F33602-88-D-0027, School of Computer Science, Carnegie-Mellon University, February 1990.

[22] Tokuda, H, J.W. Wendorf, and H.Y. Wang, *Implementation of a Time-Driven Scheduler for Real-Time Operating Systems*, Proceedings of the Real-Time Systems Symposium, IEEE, December 1987.

[23] Wang, F. and D. Mao, Worst Case Analysis for On-Line Scheduling in Real-Time Systems, Dept. of Computer and Information Sciences, Univ. of MA, June 1991.

[24] Baruah, S., G. Koren, D. Mao, B. Mishra, A. Raghunathan, L. Rosier, D. Shasha, and F. Wang, *On the Competitiveness of On-Line Real-Time Task Scheduling*, Proc. Symp. on Real-Time Systems, IEEE, December 1991.

[25] Koren, G. and D. Shasba, D^{over}: An Optimal On-Line Scheduling Algorithm for Overloaded Real-Time Systems, TR-594, Computer Science Dept., New York University, February 1992.

[26] Maynard, D.P., Understanding and Controlling Overload Behavior With Best-Effort Schedulers, Ph.D. thesis, Carnegie Mellon University, Department of Electrical and Computer Engineering, 1993 (in preparation).

[27] Peha, J.M. and F.A. Tobagi, *A Cost-Based Scheduling Algorithm to Support Integrated Services*, Proc. IEEE Infocom, April 1991.

[28] Chen, K., *A Study on the Timeliness Property in Real-Time Systems*, Real-Time Systems, September 1991.

[29] Chen, K. and P. Muhlethaler, *Two Classes of Effective Heuristics for Time-Value Function Based Scheduling*, 12th Real-Time System Symposium, 1991.

WHY SHOULD WE KEEP USING PRECAMBRIAN DESIGN APPROACHES AT THE DAWN OF THE THIRD MILLENIUM ?

Position paper prepared for the Institute for Defense Analyses Workshop on Large, Distributed, Parallel Architecture Real-Time Systems, Alexandria, VA, March 1993

Gérard Le Lann, INRIA
BP 105, 78153 Le Chesnay Cedex, France

Gerard.Le_Lann@inria.fr

1. INTRODUCTION

Too often, designers argue about merits of their favorite approach, about drawbacks of other approaches, without addressing the fundamental underlying issue :

<u>what is the rationale for adopting a particular design approach ?</u>

System design can be/should be as rigorous as demonstrations in Mathematics. There are obvious analogies shown table 1.

Too often, careful examination of published solutions/methodologies reveal logical contradictions. For example, design solutions are antagonistic with what should be the appropriate computational model (sometimes, this happens because the model is not even described !). Two examples :

- the priority ceiling protocol is non-sense in the context of distributed systems ; one of the reasons is that it rests on the assumption that processes may observe the global system state (e.g. via a global semaphore) which is known to be unfeasible in distributed systems

- the IEEE 802.4 (token-bus) LAN standard is based on a violation of the design premises ; collisions are ruled out, therefore the token-passing solution ; nevertheless, designers had to provide for collision detection and resolution -- because of token loss and virtual ring reconfiguration ; doing something equivalent in Mathematics, i.e. demonstrating a theorem that violates an axiom, is very embarassing !

Careful examination may also reveal tautologies. Consider the atomic broadcast problem. Atomic broadcast guarantees that non-faulty processors see identical histories of (possibly concurrent) broadcasts, i.e. no loss and identical orderings. The ISIS ABCAST protocol is supposed to be a solution. In fact, this protocol is incorrect if the underlying (physical) broadcast sub-system is not atomic itself ! So, what is added by ABCAST, besides overhead ?

Finally, careful examination may reveal severe design flaws. A very important example is that of distributed scheduling based on fixed (computed off-line) priorities. This naive --and erroneous-- approach is derived from the rate monotonic/deadline monotonic methodologies. These methodologies are sound and should be used in their intended context whenever appropriate, namely single processor fault-free architectures running periodic (or sporadic) tasks, characterized with simple timeliness attributes (i.e. deadlines, all tasks have the same value). In the context of such simple computational models, timeliness attributes can be rigorously transformed into time-independent (fixed) priorities, using the RM or DM methodologies.

However, there is no equivalent methodology for distributed architectures (i.e. richer, more complex, but also more realistic computational models). Hence, in the case of real-time distributed systems, designs and solutions that are based on fixed priorities are barely interesting, because the following (very difficult) issue is left open : how to transform distributed task timeliness attributes into integers so that the set of specified system-wide timing constraints are provably met ?

A real-time distributed system designer who would use fixed priority based scheduling has no means to demonstrate that he is solving the originally stated problem. Who would trust the resulting system ?

Let us be clear. We are not claiming that the rigorous design/implementation cycle shown table 1 is always feasible. There are instances of system objectives/requirements whose complexity raises design problems that are beyond the reach of current state-of-the-art in Computer Science. When faced with such problems, we are forced to use «pragmatic» approaches (e.g. heuristics, testing in lieu of proofs/validation, etc.), until state-of-the-art matures.

However, it is our view that too often, designs are poorly/erroneously conducted, not because state-of-the-art is limited but because state-of-the-art is ignored.

2. RATIONALE FOR A CORRECT COMPUTATIONAL MODEL IN THE CASE OF LDPART SYSTEMS

Our purpose is to show how one can arrive at a rigorous identification of a correct computational model intended for designing Large Distributed, Parallel Architecture Real-Time (LDPART) Systems.

We proceed by identifying the implications (noted I) of the stated system objectives/requirements.

I_1 : distribution/parallelism → concurrency
Synchronous or asynchronous concurrency ?

I_2 : large (systems) → probably asynchronous concurrency

$I_{3.1}$: real-time → fault-tolerance
Faults are stochastic events. Hence, $I_{3.1}$ combined with I_1 yields a confirmation of I_2.

⇒ Conclusion # 1 : asynchronous concurrency

In passing, let us notice the following reciprocal implications :

fault-tolerance → redundancy

fault-tolerant redundancy management → distribution/parallelism

$I_{3.2}$: real-time → scheduling
Off-line or on-line scheduling ?

Conclusion #1 precludes clairvoyance (full knowledge of the future system/environment histories such as external event arrival laws, inter-task conflicts, etc.).

⇒ Conclusion #2 : on-line scheduling.

Let us now examine the proof/validation issue. Simply stated, there are two schools of thought, or two major design approaches.

One of them, called the static design approach (S), rests on clairvoyance assumptions. Critical attributes of a LDPART system constitute a multidimensional space [external event arrival laws, task durations, fault «arrival laws», internal event (e.g. message) arrivals laws, shared resource conflict patterns, task time dependent/time independent values, etc.]. Proofs/validation established for design solutions based on clairvoyance assumptions are valid only for a particular point (at best, a small region) in this multidimensional space. In other words, the coverage factor of such proofs/validation is not very good.

The other one, called the dynamic design approach (D), is consistent with the widely accepted fact that clairvoyance assumptions are unrealistic in general. It can be demonstrated that clairvoyance assumptions are fully antagonistic with the very nature of LDPART systems. Proofs/validation can indeed be established for design solutions based on zero or limited a priori knowledge. The price incurred to establish (more complex) proofs compared with demonstrating properties under a S approach is paid only once, with the very interesting result that such proofs/validation are valid throughout the entire multidimensional space (zero a priori knowldege) or most of it (limited a priori knowledge). In other words, the coverage factor of such proofs/validation is very close or equal to one [1].

Examination of current state-of-the-art in Computer Science indicates that proofs/validation exist today for many D design solutions, the only ones to match the basic nature of LDPART systems. We believe that, as time goes by, more proofs/validation will become available for D design solutions, thus relegating some of the S design solutions to the precambrian era.

Let us give examples of design solutions that have a poor coverage factor in the context of LDPART systems. The fist example is concerned with those real-time systems that are based on the premise that external events should be looked at only when appropriate, i.e. periodically sampled, this being thought to be absolutely required to prove timeliness properties. In fact, what this really means is that such an extraordinary assumption is made to ease the designer's job ! It looks like the question of whether the environment really behaves periodically is of minor importance.

The second example is linked with the first one. A priori knowledge of periodical patterns is then used to «solve» the shared bus multiaccess problem, using a very conventional solution that is Static Synchronous Time Division Multiplexing [allocation of bus slots to (periodical) messages is pre-computed off-line]. The construction of a reliable distributed SS-TDMA bus precisely raises the fundamental issue of how to synchronize entities (bus attachment units) without resorting to centralized control. This issue does not seem to be considered worth addressing either by those who believe that it is «simple enough» to be tackled «at the hardware level». They are wrong. Synchronization is an essential algorithmic issue, even at the hardware level [2]. Now the fallacy : SS-TDMA bus based systems are publicized as «distributed systems». Those who believe in such designs fool their audience --or try do so-- and maybe fool themselves, for the most sincere of them, by calling «real-time distributed system» what turns out to be just a synchronous asymmetrical wait-state free multiprocessor. They do not realize that with this type of design, they can only mimic, with a digital technology, what used to be called analog computers !

Let us now summarize. The examination of the proofs/validation issue reveals that D design solutions have coverage factors greater than those resulting from S design solutions. This is consistent with conclusion #1, which precludes clairvoyance.

We thus conclude that the correct computational model to reason about, to design and to prove properties of LDPART systems is the model of asynchronous concurrent, on-line scheduled, computations.

3. MAJOR ISSUES

3.1. Lack of clairvoyance does not preclude proving

With our approach, the four major issues are mutually related. We will thus present how we *proceed to design and validate* LDPART *systems*. Positions and recommendations are clearly pointed out in the sequel.

Let us first mention that w.r.t. formal models and logics, we have no special recommendation, as there is no consensus on useful models of distributed/parallel systems and logics for reasoning about/proving their properties, this being especially true when considering timeliness properties. This is an example of an area where state-of-the-art does not offer sufficiently powerful methods or solutions yet.

In order to conduct a validated LDPART system design, we (not surprisingly) recommend using our general computational model (see above), which has proved to be adequate to tackle all critical issues consistently, essentially Concurrency Control (Synchronization), Real-Time Scheduling. Global Time and Fault-Tolerance. A design model, which is derived from this computational model, is used to «instantiate» those algorithms (solutions to the critical issues) that are proven correct, and appropriate for a particular LDPART system under consideration. As will be shown, this design model essentially is an object oriented transactional model. Our approach is based on proving properties of individual algorithms (solutions), on proving properties of algorithm composition, this being done within the framework given in section 2, i.e. assuming only partial knowledge or zero knowledge of the future.

We refute the conventional precambrian allegation that proving is impossible if one is not clairvoyant. Convincing arguments in favor of on-line (non clairvoyant) algorithms can also be found in [3].

Separately, when considering a particular architecture for implementation, those variables found in the (proven) solutions can be valued, thus yielding the numerical values of such measures of interest as upper bounds on response times, lower bounds on global time precision, lower bounds on task throughput, etc.

Our position is clearly to separate design and dimensioning. For those solutions selected during system design, we use established proofs of we prove properties of interest, under the form of computable functions or under the form of theorems. Let us give two examples.

The serializability theorem established for well-formed transactions which obey the 2-phase locking rule is an example of a (*safety oriented*) proof established with zero-knowledge of the future. Similarly, we have proved the existence of finite upperly bounded response times for shared multiaccess broadcast networks, in the absence of overloads, without assuming a priori knowledge of individual node traffic patterns (contrary to what is needed with SS-TDMA or token passing protocols). The corresponding protocol, called Deterministic Ethernet or CSMA-DCR (Deterministic Collision Resolution), has a variant called DOD-CSMA/CD (Deadline Oriented Deterministic CSMA/CD) which is capable of handling messages that have deadlines. We have proved that DOD-CSMA/CD is an optimal on-line distributed scheduling algorithm for periodic and sporadic message arrivals. This, and the absence of overload, are the only clairvoyance assumptions needed. We conjecture that an optimality proof can also be given in the case of overloads and for aperiodical arrivals.

3.2. A design model for LDPART systems

Execution of asynchronous concurrent computations is conducted via Concurrency Control (CC) algorithms. We rely on the Serializability Theory and on related CC algorithms to prove properties (safety and liveness properties) of concurrently executing application S/W modules. A real-world, well established, incarnation of the Serializability Theory is the Transactional Model, which turns out to match Object Orientation quite well. We have found the following design model to be very appropriate when considering LDPART systems :

- a transaction is a set of actions/threads that invoke/enter objects (to access computational resources)
- the binding between actions/threads and objects is dynamic (a consequence of our rejection of clairvoyance)
- objects are characterized with three attributes, namely multiplicity, persistency, access.

Multiplicity

- *unique instantiation (UI) of every object, no assignment problem*
- multiple instantiations (MI) of every object, constituting a class per object (e.g. processors, multiple copies of a variable), raising an assignment problem

Persistency

- no persistency property (NP), for such objects as processors, communication links
- persistency property (P), for such objects as mechanical devices (e.g. physical orientation of a robot arm), data structures

Access

- centralized (C), i.e. global knowledge (GK) is available (within the limits of what is possible when assuming no clairvoyance)
- distributed (D), i.e. only partial or incomplete knowledge (subsets of GK) is available.

System-wise, there are two possibilities :

- single-object systems (e.g. a uniprocessor, a communication channel) (SOS)
- systems comprising multiple (non equivalent) objects (MOS)

No need to elaborate on the well-known advantages of the Transactional Model and Object-Orientation w.r.t. S/W engineering as well as w.r.t. the existence of guaranteed properties, such as ACID properties for transactions and on-line updatable data (All-or-nothing, Consistency, Isolation, Durability).

On the contrary, it might be necessary to elaborate on the following. LDPART systems contain and maintain distributed data structures that are updated on-line. Examples are sensor/actuator status tables, environmental data, internal system tables. Hence the need to use CC algorithms.

Of course, safety or liveness properties are not equivalent to timeliness properties. Consequently, only those CC algorithms that can be composed with on-line schedulers should be considered.

3.3. Distributed control of asynchronous concurrency and distributed on-line scheduling

The design model given above can now be used to reason about such a composition. Table 2 shows a condensed taxonomy-oriented view of the issues, for the UI case (essentially, the MI case is obtained by adding the assignment problem). Table 2 can be illustrated as follows. (HPF stands for Highest Priority First, EDF stands for Earliest Deadline First).

c_1	=	HPF/EDF for uniprocessors
c_2	=	HPF with priority inheritance for uniprocessors
c_3	=	scheduling over asymmetrical shared-memory multiprocessors
c_4	=	HPF and priority ceiling over shared-memory multiprocessors
d_1	=	EDF over a broadcast communication channel
d_2	=	EDF over a multiclient/single file server system
d_3	=	best-effort scheduling over a distributed system
d_4	=	EDF + 2-phase locking + deadline-based deadlock avoidance

It is now easy to address the combined issues of CC and Scheduling. Under a deadlock prevention approach, the CC issue disappears. One is left with the problem of choosing an optimal on-line distributed scheduling algorithm. Under a deadlock avoidance or a deadlock detection-resolution approach, one must select CC and Scheduling algorithms in such a way that, for any given conflict scenario between two real-time transactions A and B, both algorithms make the same decision (A precedes B or vice-versa). The example given for d_4 illustrates a compatible combination. However, this correct combination is not optimal, i.e. it cannot handle some scenarios (e.g. overloads) that can be accommodated by other combinations.

It might be appropriate to concentrate a little on the notion of optimality for on-line distributed algorithms. This notion is built upon the definition of optimality for on-line algorithms. An optimal on-line algorithm has the smallest achievable competitive ratio, noted r (i.e. the smallest loss compared to a clairvoyant algorithm). Given the same amount of limited a priori knowledge, no other algorithm can dominate an optimal algorithm.

When distribution must be accounted for, global knowledge GK that is given a priori is not accessible to an algorithm (as is the case with centralized on-line algorithms). Consequently, some (distributed) algorithm must be used to «bridge the gap», i.e. to approximate GK. Assume, for example, that broadcasting is free of charge and instantaneous. Then, distributed schedulers are always cognizant of those tasks waiting to be scheduled, system-wide. In this (ideal) case, «distributed» optimality is equivalent to «centralized» optimality.

Optimality in distributed systems corresponds to achieving the best approximation of GK, noted k, i.e. the smallest uncertainty ratio u = GK/k. Consequently, an optimal on-line distributed algorithm has the smallest achievable competitive ratio u*r.

As mentioned before, DOD-CSMA/CD is an example of a provably optimal on-line distributed scheduling algorithm for case d_1 (table 2), for the computational model indicated.

Case d_1 (as well as case d_2) complexity is relatively easily tractable. This is not the case with cases d_3 and d_4. Some satisfactory solutions have emerged very recently. However, the design and validation of LDPART systems will not be a mature discipline until more correct, and possibly provably optimal, solutions become available for case d_4.

Also, the concept of competitive ratio should be refined, so as to yield more realistic bounds than those established assuming an all-knowing adversary, which is overly pessimistic in many real world settings.

Let us conclude this section by briefly showing what can be achieved under our approach. Let us use the DOD-CSMA/CD example again. Without any a priori knowledge, it is possible to prove that under worst-case conditions (a global collision among n contenders), it takes at most T to transmit all messages successfully, with

$$T = \left[\left\lceil \frac{n}{2} \right\rceil - 1 + \left\lfloor \frac{n}{2} \right\rfloor (2\log_2 F - 1) \right] s + \alpha_n ,$$

where F is the number of equivalence classes chosen for deadlines (i.e. comparable deadlines), s is the conventional CSMA/CD channel slo: time and α_n represents the total message (n of them) transmission time (without collisions).

It is possible to obtain bounds also for the algorithms we have selected to solve the (more difficult) combined problem of distributed Scheduling and multiple-object Concurrency Control.

Validation of a design is then very easy. We use the parameters given to characterize a particular LDPART system (e.g. arrival laws, individual task/message durations). From the analytic expressions of bounds, we compute the corresponding numerical values and check whether they match the specified system-wide requirements/objectives. If not the case, a different design solution must be considered or additional a priori knowledge must be provided (although this will reduce the coverage factor of the corresponding solutions).

3.4. Fault-tolerance

With LDPART systems, many abnormal behaviors (called failures) could result from multiplexing together an a priori unknown number of application S/W modules (transactions) over an a priori unknown number of computational resources. Given our proof-based approach and our computational model, such failures cannot exist, if it can be assumed that :

(i) every individual transaction is correctly implemented,

(ii) every algorithm is correctly implemented.

We are thus restricting the scope of fault-tolerance in LDPART systems to the familiar area of conventional S/W engineering and need only draw solutions from current state-of-the-art in Fault Tolerance.

We will not discuss here the relative merits and drawbacks of S/W engineering solutions directed at producing dependable S/W. Whatever solution is used, operational S/W still contains «bugs». The best (non speculative) solutions to this problem today are those used in Tandem systems (process pairs, primary/backup, checkpoints) and in Stratus systems (process pairs, dual duplex processing). Although process replication does not solve the «solid bug» problem (S/W design faults), experience indicates (statistics published by Tandem Corp. in particular) that process replication solves most H/W failure related problems as well as the «transient bug» problem, which seems to largely dominate the «solid bug» one.

Replication of process execution at run-time and replication of H/W as well as of system S/W (OS) need to be hidden. It is no coincidence that the Transactional/Object Orientation Model has proved itself as being well suited to provide application S/W developers with a programming interface that fully hides the intricacies of the solutions used to achieve a given degree of system-wide fault-tolerance. This is not surprising indeed, as the same design Model has proved itself to be well suited also to hide the intricacies of the Distributed Concurrency Control and Scheduling solutions, as indicated above.

Regarding fault-tolerance, from a more general (and maybe theoretical) viewpoint, one must be aware of the fact that the «Real-Time community» implicitly relies on what is inappropriately called a «synchronous» computational model (i.e. a model where upper bounds on computation/ communication delays are known a priori). Fault-tolerance under such a model might raise the need to perform on-line assumption verification (that the bounds are not violated). This is a delicate issue. We recommend that snecial attention be paid to this issue. Especially when systems are «large», assumptions on a priori knowledge of «good» bounds might have a bad coverage factor. Hence, it might be necessary to use a «partially synchronous» computational model (i.e. bounds exist but are not known a priori) which still permits deterministic solutions, or an «asynchronous» computational model (i.e. bounds do not exist), where only probabilistic or randomized solutions can be contemplated.

A last point might be worth addressing, that is testing/debugging of LDPART systems. As we still do not know how to produce fault-free S/W, we still have to rely on testing/debugging. It is often heard that this task is rendered more difficult in the case of distributed architectures. We do not understand this statement. We suspect that it might result from a lack of understanding of what is needed for a real-time distributed system to run correctly. We have shown that CC algorithms are needed, in particular to enforce particular orderings or histories on sets of concurrent events. Typically, CC algorithms use attributes representing some notion of logical time, such as timestamps or tickets. These attributes reflect causality relationships among events. Similarly, global physical time must be maintained in a LDPART system. Physical timestamps reflect chronological relationships among events.

It is therefore possible to record separately histories or traces as produced by individual components of a LDPART system and merge them consistently by using the logical/physical attributes associated to events. This yields the possibility of observing system-wide causally and chronologically correct histories, as is the case with conventional (centralized) systems.

4. CONCLUSION

We have developed a number of arguments which, we hope, should help our community to save time and invest resources where appropriate. LDPART systems are raising issues which cannot be correctly tackled without knowledge of current state-of-the-art in Computer Science. Even though very simple designs have been satisfactory in the past, that will be less and less often the case. The precambrian era is over. Given the continuing advances made in H/W, we feel it mandatory to adopt an open-minded approach to the challenging issues raised with LDPART systems, so as to avoid facing somewhat embarassing conclusions such as not allowing oneself to use off-the-shelf technology (e.g. cache-memory based computers) because they do not fit an excessively poor/ naive computational model.

References

[1] G. Le Lann, «Designing real-time dependable distributed systems», Computer Communications, (Butterworth-Heinemann pub.), vol. 15, n° 4, May 1992, pp. 225-234.

[2] A.J. Martin, «Tomorrow's digital hardware will be asynchronous and verified», IFIP Congress 92, (Elsevier North-Holland pub.), Madrid, September 1992, pp. 684-695.

[3] R.M. Karp, «On-line algorithms versus off-line algorithms : how much is it worth to know the future ?», IFIP Congress 92, (Elsevier North-Holland pub.), Madrid, September 1992, pp. 416-429.

Computer Science		Mathematics
system objectives/requirements	≡	disciplines
↓		↓
computational model	≡	axioms
↓		↓
design solutions	≡	theorems
↓		↓
implementation	≡	particular application of theorems

Table 1: a rigorous design/implementation cycle

	UI			
	SOS		MOS	
NP	c_1	d_1	c_3	d_3
P	c_2	d_2	c_4	d_4
	C	D	C	D

c_1 : conventional Scheduling (single processor)

c_2 : conventional Scheduling with critical sections

c_3 : conventional Scheduling (+ broadcasting/dispatching), no CC

c_4 : same as c_3, with centralized multiple-object CC

d_1 : distributed conventional Scheduling (single distributed object), no CC

d_2 : same as d_1, single-object CC

d_3 : distributed Scheduling, no CC

d_4 : same as d_3, multiple-object CC

Table 2: Concurrency Control and Scheduling

ISSUES IN DISTRIBUTED REAL-TIME SYSTEMS

Jane W. S. Liu

Department of Computer Science
University of Illinois
Urbana, Illinois 61801

INTRODUCTION

In recent years, significant progress has been made in the design and evaluation scheduling algorithms that are basic building blocks of real-time systems containing one or a few processors and supporting traditional embedded applications. In particular, the rate-monotonic approach [1-4] to scheduling real-time computations and data communications is well developed. Algorithms for assigning periodic tasks to processors, processor scheduling, I/O bus and broadcast network access, synchronization between periodic tasks, and interrupt handling are now available (e.g., [5-8]). There is a growing collection of rigorous performance bounds and simulation/measurement data to support design choices and decisions. Systematic design and synthesis methods and sound validation and testing strategies are beginning to emerge.

In contrast, the basic methodologies needed to build and validate distributed real-time systems are not yet available. This is especially true when the applications are complex and dynamic and have critical timing constraints. Many problems on scheduling and resource access control of distributed real-time applications remain to be solved. Systematic design and synthesis methods and tools need to be built on not only the solutions to these problems but also rigorously established boundaries governing the correct and safe usage of the solutions. Examples of key problems include how to schedule tasks to meet their end-to-end timing constraints in distributed and parallel environments; how to make scheduling and resource access control decisions when information needed to support such decisions are old, partial or unavailable; how to predict and prevent oscillatory behavior and instability of the resultant systems built on dynamic scheduling strategies; and how to ensure graceful degradation in the presence of overloads and failures.

This paper first describes a view of distributed and parallel real-time systems and suggests a hierarchical approach to scheduling and resource management and to reasoning about the timing behavior of large distributed systems. It then discusses the key problems mentioned above and concludes by giving additional arguments on why the design of real-time systems should have sound theoretical foundations in real-time scheduling.

MODELS OF LARGE, DISTRIBUTED, AND PARALLEL SYSTEMS

By a large system, we mean one that contains hundreds or thousands of tasks. Here, the terms *tasks* and *subtasks* loosely refer to individual units of work that are allocated resources and executed to support the functionalities of the system. For example, a task or a subtask may be a granule of computation, a unit of data transmission, a response to a query, or even an operator action. It executes on a computer, a data link, a database, or an operator console. We model all resources on which tasks execute abstractly as *processors*.

A system may contains many different *types* of processors. Processors of different types cannot be used interchangeable either because they are functionally different (e.g., signal processors, data processors, and communication links are functionally different processors), or because they are located at different places (e.g., the data links connected to onboard sensors are distinct from the links to a decision support system on the ground.) On the other hand, processors of the same type are considered to be identical or compatible because they can be used interchangeably.

Concurrency

In a complex distributed system, a task typically consists of subtasks that are dependent on each other and execute in turn. For example, in an air traffic control system, the sequence of operations that is carried out when an aircraft first enters a coverage area is such a task. It consists of a sequence of subtasks which model the processing of the radar sensor data on a signal processor to generate a track record, the transmission of the track record to a data processor, the correlation of the track record with other records on the data processor, the transmission of the aircraft characteristics from the data processor to an intelligent decision support system, the analysis by the decision support system, and so on, until the correct action is taken. The processors on which the subtasks execute are different because they support different functions. On the other hand, the processors that handle the displays in an airport control tower and an enroute control center may be functionally identical. We nevertheless consider them to be of different types because we do not want to execute the subtask that updates one display on the processor directly connected to the other display under normal operating conditions. Similarly, in a command, control and communication system, the task of sending a message can be decomposed into subtasks that route and transmit the message from the sender to the receiver(s) in a large network of data links and switches. The switch and link connected to the sender are different from the switch(es) and link(s) connected to the receiver(s) because they cannot be used interchangeably.

From the examples above, we see that concurrency in a distributed system arises naturally as subtasks of different tasks sequencing through different processors in a pipeline fashion. This type of concurrency can be captured to a great extent by the classical *job shop* and *flow shop* models and their variations [9-12]. The former models a system in which the subtasks in different tasks execute on different processors in arbitrary orders, while the latter models a system in which the subtasks in different tasks execute on different processors in the same order.

A variation of the classical model is the *constrained job shop*; this model characterizes a distributed system as a set of heterogeneous flow shops that share processors. In other words, a system contains many classes of tasks: tasks in the same class execute on different processors in the same order, but tasks in different classes execute on different processors in different orders. The corresponding queuing theoretical model is a network of queues with many job classes and multiple routing chains. Each chain gives the order in which tasks in a class execute on different processors. An example where this model is appropriate is a multihop, real-time network. The individual flow shops model virtual-circuit connections established in the network. Each connection carries multiple streams of real-time data that must be delivered from one end of the connection to the other end in time.

Other variations of the classical flow shop and job shop models are *flow shop with recurrence* and *periodic flow shop* and job shop [12]. In a flow shop with recurrence, each task executes more than once on one or more processors. This variation models a system that does not have a dedicated processor for every function. An example is a control system containing three computers: a sensor data processor, a computation server, and an actuator command generator. The computers are connected by a token ring. We can model the token ring as a processor and the system as a flow shop with recurrence. Each task executes first on the sensor data processor, then on the ring, on the computation server, on the ring again, and finally on the command generator. In a periodic flow shop or job shop, each task, and hence each subtask, is a periodic sequence of requests for the same computation or communication. A multi-hop connection that is used to transmit periodic multimedia data and a distributed air traffic control system that periodically processes radar returns, tracks aircrafts and displays their flight paths can be modeled as periodic flow shops or jobs shops.

Parallelism

In addition to concurrent executions of subtasks on different types of processors, parallel executions are feasible whenever there are more than one processor of same type. In an air traffic control system, for example, there may be an array of signal processors, making it possible to execute many signal processing subtasks of different tasks in parallel. Similarly, multiple links and switches between the sender and the receiver(s) provide parallel paths that can be used to increase throughput or reduce message delay. The traditional multiprocessor and redundant-processor models (e.g. [13-18]) used in studies on parallel and distributed scheduling, task assignment and load balancing capture this type of parallelism. Such a model characterizes a subsystem abstractly as a set of processors that are either identical or compatible; a subtask can execute on any of them. Each subtask may be further divided into subtasks of smaller granularity so that the degree of parallelism can be increased with possible accompanied increases in communication and scheduling overheads.

Some processors may in fact be massively parallel machines. Examples of such processors include some parallel systems designed to do image enhancement, feature extraction and object identification. Fine-grain parallelism can be exploited to speed up the completion of subtasks only when we have good parallel computation algorithms and parallelizing compilers. The issue here is not about real-time scheduling.

REAL-TIME CONSTRAINTS

By a *real-time system* here, we mean specifically a computing and communication system in which a significant number of of tasks have critical timing constraints. Timing constraints of a task can almost always be expressed in terms of its (absolute or relative) *release time* and *deadline*. The former is the time instant after which it can begin execution. The latter is the time instant by which it must be completed. A *timing fault* is said to occur when one or more timing constraints are violated. A real-time system operates correctly only in the absence of timing faults.

Timing constraints that follow naturally from high-level requirements of distributed real-time applications are typically end-to-end in nature. For example, in a collision detection and avoidance system, the maximum allowed elapse of time from the instant when a target is detected to the instant when an evasive action must be taken is determined by how soon a collision can occur. The requirement that a correct evasive action is taken in time imposes an overall deadline on the task consisting of a sequence of computation and communication subtasks. (These subtasks process radar returns, identify the target, compute its course, choose the evasive action, and generate and send the commands to the actuators.) As long as the sequence is completed by the deadline, it is not important when the intermediate computational subtasks are done or how long messages are delayed.

In other words, to meet end-to-end timing constraints of a task, we are required to begin executing its first subtask at or after its release time and complete the execution of its last subtask by its deadline. The intermediate subtasks have only derived release times and deadlines; their executions are constrained only by the dependencies between them and by the fact that they must be completed sufficiently early to allow the on-time completion of the last subtask. Their lack of application-imposed timing constraints provides the system with more freedom in scheduling the intermediate subtasks. This freedom, together with our desire to take advantage it to achieve greater efficiency in resource usage, increases the complexity of scheduling and resource access control in distributed and parallel environments.

HIERARCHICAL APPROACH

The characteristics of concurrency and parallelism in task executions and the end-to-end nature of their timing requirements suggest a hierarchical approach to scheduling in distributed environments. The primary goal of scheduling and resource access control in a distributed real-time system should be to enforce the end-to-end timing constraints that directly follow from high-level timing requirements of the applications supported by the system. In end-to-end scheduling of time-critical tasks, we want to assign intermediate release times and deadlines to their subtasks and to schedule the subtasks on the individual processors so as to ensure the completion of all tasks in time whenever it is feasible to do so. We want the resultant system to be responsive to varying demands, to degrade gracefully in the presence of overloads and failures, and to be easy-to-modify, maintain, validate and test.

Scheduling subtasks on interchangeable processors have a set of secondary goals, including to make good use of parallelism, to maximize the likelihood of on time completion of subtasks, to equalize resource utilizations, to provide redundancy, to increase availability, etc. The traditional focus of research on distributed and parallel systems has been on ways to meet these secondary goals. Past works on parallel and distributed scheduling have produced many excellent task assignment and load balancing schemes, as well as performance bounds of multiprocessor scheduling algorithms; examples of these results can be found in [13-19]. These schemes can enhance end-to-end scheduling algorithms in order to improve the overall robustness, efficiency, and availability of a distributed system but, by themselves, are not solutions to the end-to-end scheduling problem.

To illustrate the relation between the end-to-end scheduling problem and the traditional distributed scheduling problems, we note that one way to do end-to-end scheduling is to assign intermediate release times and deadlines to all subtasks. Af ›r the intermediate release times and deadlines are assigned, we can then use some task assignment, multiprocessor scheduling, load balancing and task migration schemes to schedule the subtasks on processors of each type, trying to make the best use of parallel resources. Therefore, an end-to-end scheduler can be viewed as a high-level system module tha' contains multiprocessor schedulers, load balancers and resource managers as components.

END-TO-END SCHEDULING ISSUES

We now examine several issues in end-to-end scheduling that remain to be addressed. They are concerned with increasingly more complex and dynamic situations, listed here according to the amount of load and status information available to support scheduling decisions and the costs of maintaining this information.

Scheduling with Global Information

Almost all available end-to-end scheduling algorithms that are supported by rigorous performance bounds and profiles are suited only for systems and subsystems that are either sufficiently small and tightly-coupled or sufficiently static. Examples of the former include flight control and radar signal processing systems, which are really multiprocessor systems. Examples of the latter include industrial process control, multihop virtual-circuit networks, and flight management systems under their normal operating conditions. In these systems, it is feasible to collect and distribute global load and status information and keep the information sufficiently current. Consequently, it is reasonable to assume that the scheduler for each processor, or each type of processors, knows the timing and resource requirements of all the tasks in the system as well as the decisions made by other schedulers. Therefore, it is feasible for the schedulers to work closely together and arrive at compatible decisions, or even do scheduling centrally.

Again, a reasonably realistic workload model for systems on which global status information is available is the constrained job shop model. Rather than scheduling all the tasks from all classes together according to one algorithm, a practical strategy is to partition the time available on processors of each type and allocate them to task classes. The processor time allocations are adjusted on an infrequent basis, (during mode changes, new connection establishments, etc. for example). This allows the tasks in each class to be scheduled according to a scheduling algorithm suited for the class. A system that contains N classes of tasks and uses such a semi-static partition and allocation strategy can be modeled as a system of N flow shops. Many known algorithms designed for scheduling in flow shops and job shops to meet end-to-end deadlines or to minimize lateness and algorithms for scheduling jobs in factories (e.g. [12,19,20]) are likely to be applicable. These algorithms should be studied and thoroughly evaluated.

Algorithms for assigning intermediate release times and deadlines to subtasks in periodic flow shops and job shops are also emerging. One way assumes the use of a preemptive static-priority-driven algorithm (e.g. the rate-monotonic or deadline-monotonic algorithm) to schedule subtasks on each type of processors. In this case, the rate-monotonic (or deadline-monotonic) technique can be extended straightforwardly to deal with end-to-end scheduling. Another example of periodic job shop scheduling algorithms uses a novel convex programming method and an iterative modification method to divide the feasible interval between the release time and deadline of every task into segments in order to assign intermediate deadlines, to check whether schedulability conditions on all processors are met, and to modify the intermediate deadlines if necessary.

How to integrate task assignment, load balancing and concurrency control functions into end-to-end scheduling is a problem that has not yet received much attention. Sound strategies need to be developed. Strategies and algorithms assuming current global information can serve as benchmarks against which we can measure the effectiveness of algorithms that do not rely on the global status information.

Scheduling Based on Local and Global Information

In large and/or dynamic systems the information about the global system state is not always available and is usually too costly to keep current. This case of the end-to-end scheduling problem resembles the problems in routing and flow control in packet-switched networks. The major difference between the problems is in the primary objectives of scheduling: we want to make sure that all time-critical tasks will meet their end-to-end deadlines while the objectives of routing and flow control are to keep the average end-to-end response time low and the overall throughput high, and to ensure fairness among tasks. Nevertheless, there are good lessons to be learned from strategies in routing and flow control for coping with missing or old global information.

Using Periodically Updated Global Information — For example, the strategy used in a version of the arpanet routing algorithm is to let each switch make its routing decisions based on its own information on the global load condition. This information is updated periodically. The rationale behind this strategy is that the information will remain sufficiently current throughout the update period and the routes chosen based on this information will be sufficiently good. A similar strategy for end-to-end scheduling is feasible when the system consists mostly of periodic tasks and sporadic tasks with bounded interarrival rates. The number of subtasks on each processor (type) and their total utilization can be made available periodically to all schedulers in the system, for instance. The scheduler for each processor can use this information to estimate when new subtasks are likely arrive. This makes it possible for the scheduler to do some pre-planning, to increase the schedulability and reduce the completion times of its subtasks. Design parameters of this strategy are the amount of information exchanged by the schedulers and the length of the update periods. How to choose these parameters based on the time parameters and dynamics of the task system is a question to be answered.

Using Local Information Only — Rather than scheduling the executions of all the subtasks in each task in a coordinated manner as discussed earlier, an opposite approach is to have the scheduler on each processor decide when to execute what subtasks based on the information it has about the parameters of its own subtasks alone. No information is exchanged between schedulers. In this case, a scheduler cannot predict the arrivals of new subtasks and their parameters. Moreover, the execution times of later subtasks may be unknown; the given end-to-end deadlines do not provide much information on when the intermediate subtasks must be completed. There are many reasons to believe that we will have poor performance when scheduling decisions are based solely on local information. Under the assumptions stated above, there is no choice but to schedule the subtasks on each processor completely on-line. It is known that the performance of completely on-line scheduling is poor [21]. We can also view an end-to-end schedule produced independently by the individual schedulers as a priority-driven schedule based on decisions that are at best locally optimal. It is known that priority-driven scheduling strategies have unacceptably poor worst-case performance in systems that contain functionally dedicated processors [16]. In despite of all the preliminary evidence against it, local scheduling approach should be thoroughly evaluated for two reasons. First, this approach is the only feasible one in extreme situations when no prior knowledge about tasks parameters are available; even when any task will be released and ready for execution is unknown. Second, its performance data will give us a set of benchmarks on one end of the performance spectrum, with the data on algorithms for scheduling with complete knowledge on the other end.

Using Both Local and Global Information — The strategies that use both local and global information and combine local decisions with global decisions are likely to be effective. In particular, local information that is current can be used to supplement the global information that becomes old. In this way, we can reduce the update frequency of the global information and improve the responsiveness of the system. A well-known example of strategies for combining global and local decisions is delta routing in networks [22]. Analogous strategies for combining global information with local information to make end-to-end scheduling more responsive and robust while keeping the cost of maintaining status information low should be explored. Specifically, hybrid scheduling strategies combine off-line scheduling based on global information with on-line scheduling based on local information. The global schedule(s) is modified infrequently. Some dynamic conditions have short time constants, making it impossi^ble or too costly to maintain global information about them. Responding to these conditions is taken care by local schedulers. They try to schedule on-line the unexpected tasks as best as they can based on their local information and the guideline given by the global schedule. These strategies fit well in the framework of the rate-monotone paradigm and the imprecise computation paradigm [24-27].

Dynamic, Adaptive and Monitor-Based Scheduling

Algorithms known as dynamic algorithms, adaptive algorithms and monitor-based algorithms assume that the system condition is monitored and scheduling decisions are based on the observed condition. (We will simply refer to them all as adaptive algorithms.) In some cases, decision rules may also change as the observed condition changes.

It is well known that adaptiveness can lead to oscillatory behavior and instability. We cannot use a adaptive algorithm for scheduling functionally critical tasks until we thoroughly understand the dynamic behavior of the resultant system and have effective methods for predicting oscillatory behavior and instability and for preventing them. One way to prevent instability and oscillatory behavior is to make adaptive schedulers less sensitive to changes in the system condition. Another way is to avoid synchronous reactive actions of all the schedulers. Unfortunately, all the effective methods tend to make the system less responsive and impose a limit on how dynamic a system can be. It is important for us to know this limitation.

Enhancing Graceful Degradation

The imprecise computation technique [24-27] is a natural way to provide graceful degradation and to cope with uncertainty in system load. We call a system based on this technique an *imprecise system*. In an imprecise system, each time-critical task is structured in such a way that it can be logically decomposed into two portions: a mandatory portion and an optional portion. The mandatory portion of the task must be completed to produce an approximate result of an acceptable quality. This portion of a time-critical task must be completed by the deadline of the task. The optional portion refines the approximate result produced by the mandatory portion. We can choose to leave this portion unfinished and terminated prematurely if necessary, with an accompanied reduction in the result quality.

An example of where the imprecise computation technique is applicable is facsimile transmission. When the progressive built-up method is used, the data encoding each still image is divided into four blocks; each additional block gives a clearer image. The mandatory portion is the transmission of the first one or two of the four blocks that gives a fuzzy but intelligible image. Other blocks can be discarded when the network is congested. Other examples include transmissions of compressed voice and video, tracking and feedback control. In each case, we often prefer to have a timely result of a poorer quality than a late result of the desired quality.

The imprecise computation technique makes meeting all timing constraints significantly easier for the following reason. To ensure that all deadlines are met, we only need to ensure that all the mandatory portions of all tasks are completed by their deadlines. This can be done by restricting all the mandatory portions to have bounded execution time and resource requirements and scheduling them in a conservative and robust way. The leftover system resources can be used to complete as many optional portions as possible. It is not necessary to eliminate non-determinism in the timing and resource requirements of optional subtasks, thus allowing greater freedom in their design and implementation. For different types of applications, the costs and benefits in the tradeoff between the result quality and execution time requirements are more appropriately measured by different criteria. Many scheduling algorithms have been developed to tradeoff according to these criteria, including the ones described in [24-26]. Among the existing algorithms for scheduling imprecise computations, many can be modified for end-to-end scheduling. Examples include the class of algorithms for scheduling periodic subtasks and for on-line scheduling. The imprecise computation approach to flow and congestion control is feasible for voice, video and other messages as long as they are encoded using a technique that allows their transmissions to be imprecise.

SUMMARY

The hierarchical approach to scheduling and resource access control points to a hierarchical decomposition approach to design, synthesize and test large real-time systems: A large system is decomposed into subsystems. From scheduling point of view, each subsystem consists of subtasks that execute on processors of the same type. The timing requirements of the individual subsystems are derived from the timing requirements of the system as a whole. To ensure that all timing constraints of the system are met involves making sure that the timing constraints of each subsystem are met and that the end-to-end scheduling strategy robustly enforces the overall timing constraints as it integrates the subsystems together. The hierarchical decomposition approach is in fact old and commonly used. Its effectiveness in dealing with large distributed real-time systems is being questioned primarily for the following reason. The intermediate timing constraints of subsystems are typically assigned in a trial-and-error manner, and the end-to-end timing constraints are often not met when the subsystems designed to meet their assigned timing constraints are scheduled together. What we need to make this approach effective are principles in end-to-end scheduling and resource allocation that can guide the decomposition and integration processes.

Past experience has led us to believe that a large distributed system with critical timing constraints must be designed and built on sound and rigorous scheduling theories. Such a system has an intractably large number of states. It will be impractical, or even impossible, to validate and test the system in an enumerative and exhaustive manner. We need non-exhaustive validation and testing strategies that have provably complete coverage. A hierarchical approach to scheduling and resource access control should lead to system architectures for which such strategies are likely to be feasible.

Many distributed systems built to date are known to exhibit nondeterministic, oscillatory and unstable behaviors. Because it is difficult to produce the conditions under which the system behaves in such an undesirable manner during testing, the fact that they can occur is often discovered when the system has been put in use and failed unexpectedly. We need scheduling theory to provide us with not only algorithm and protocols to map tasks to processors and resources, but also insights and thorough understanding of the resultant system so that we can predict and eliminate the oscillatory and unstable conditions.

REFERENCES

[1] Liu, C. L. and J. W. Layland, "Scheduling algorithms for multiprogramming in a hard-real-time environment," *Journal of the Association for Computing Machinery, Vol. 20, No. 1*, pp. 46-61, January 1973.

[2] Dhall, S. K. and C. L. Liu, "On a real-time scheduling problem," *Operations Research*, Vol. 26, No. 1, pp. 127-140, February 1978.

[3] Lehoczky, J., L. Sha, J, K. Strosnider, and H. Tokuda, "Fixed priority scheduling theory for hard real-time systems," pp. 1-30, in *Foundations of Real-Time Computing: Scheduling and Resource Management*, Edited by A. M. Van Tilborg and G. M. Koob, Kluwer Academic Publishers, 1991.

[4] Sha, L., M. H. Klein and J. B. Goodenough, "Rate-monotone analysis for real-time systems," pp. 129-156, in *Foundations of Real-Time Computing: Scheduling and Resource Management*, Edited by A. M. Van Tilborg and G. M. Koob, Kluwer Academic Publishers, 1991.

[5] Sprunt, B., L. Sha, and J. P. Lehoczky, "Aperiodic task scheduling for hard real-time systems," *Journal of Real-Time Systems*, 1, pp. 27-60, 1989.

[6] Argawal, G., B. Chen, W. Zhao, and S. Davavi, "Guaranteeing synchronous message deadlines with timed-token protocol," *Proceedings of the 12th International Conference on Distributed Computing Systems, Yokohama, Japan, June 1992.*

[7] Rajkumar, R., L. Sha, and J. P. Lehoczky, "Real-time synchronization protocols for multiprocessor systems," *Proceedings of the Real-Time Systems Symposium*, Huntsville, AL, December 1988.

[8] Baker, T., "A stack-based resource allocation policy for real-time processes," *Proceedings of the Real-Time Systems Symposium*, Orlando, Florida, December 1990.

[9] Garey, M. R., D. S. Johnson, and R. Sethi, "The complexity of flow shop and jobshop scheduling," *Math. Oper. Res.*, vol. 1, 1976.

[10] French, S., *Sequencing and Scheduling: An Introduction to the Mathematics of the Job-Shop*, Wiley 1982.

[11] Grabowski, J. E. Skubalska, and C. Smutnicki, "On flow shop scheduling with release and due dates to minimize maximum lateness," *J. Oper. Res. Soc.*,, Vol 34, 1983

[12] Bettati, R. and J. W. S. Liu, "End-to-end scheduling to meet deadlines in distributed systems," *Proceedings of the 11th International Conference on Distributed Computing Systems*, Yokohama, Japan, June 1992.

[13] Eager, D. L., E. D. Lazowska, and J. Zahorjan, "Adaptive load sharing in homogeneous distributed systems," *IEEE Transactions on Software Engineering*, Vol. SE-12, No. 5, may 1986.

[14] Mirchandaney, R., D. Towsley, and J. Stankovic, "Adaptive load sharing heterogeneous systems, *Proceedings of the 9th International Conference on Distributed Computing Systems*, 1989.

[15] Yu, P. S., A. Leff, and Y. H. Lee, "On robust transaction routing and load sharing," *ACM Transactions on Database Systems*, Vol. 16, No. 3, September 1991.

[16] Liu, J. W. S. and C. L. Liu, "Performance analysis of multiprocessor systems containing functional dedicated processors," *Acta Informatica, Vol. 10*, pp. 95-104, 1978.

[17] Blaswicz, J., "Selected topics in scheduling theory," *Annals of Discrete Mathematica*, Vol. 31, 1987.

[18] Lawler, E. L., J. K. Lenstra, A. H. G. Rinnooy Kan, and D. B. Shmoys, "Sequencing and scheduling: algorithms and complexity," Centre for Mathematics and Computer Science, Amsterdam, 1989.

[19] Adams, J., E. Balas, and D. Zawack, "The shifting bottleneck procedure for job-shop scheduling," *Management Science*, Vol. 34, No. 3, 1988.

[20] Sycara, K. P., S. F. Roth, N. Sadah, and M. S. Fox, "Resource allocation in distributed factory scheduling," IEEE *Expert*, February 1991.

[21] Baruah, S., et al, "On the competitiveness of on-line real-time task scheduling," *Proceedings of IEEE Real-Time Systems Symposium*, San Antonio, TX, December 1991.

[22] Rudin, H., "On routing and delta routing: a taxonomy and performance comparison of techniques for packet-switched networks," *IEEE Transactions on Comm*, Vol. COM-24, pp. 43-59, January 1976.

[23] Chung, J. Y., J. W. S. Liu, and K. J. Lin, "Scheduling periodic jobs that allow imprecise results," *IEEE Transactions on Computer*, Vol. 39, No. 9, pp. 1156-1174, September 1990.

[24] Liu, J. W. S., K. J. Lin, W. K. Shih, A. C. Yu, J. Y. Chung, and W. Zhao, "Algorithms for scheduling imprecise computations," *IEEE Computer*, Special Issue on Real-Time Systems, May 1991.

[25] Shih, W. K. and J. W. S. Liu, "On-line scheduling of imprecise computations to minimize total error," *Proceedings of the 13th IEEE Real-Time Systems Symposium*, Phoenix, Arizona, December 1992.

[26] Cheong, I., "Scheduling error-cumulative periodic tasks," Ph.D. thesis in preparation, Department of Computer Science, University of Illinois, August 1992.

[27] Liu, J. W. S., K. J. Lin and C. L. Liu, "Imprecise computations: a means to provide scheduling flexibility and enhance dependability," to appear in *Readings on Real-Time Systems*, Edited by Y. Lee and M. Krishna, IEEE Press.

On Software Architecture for Large, Distributed, Parallel Real-Time Systems

C. Douglass Locke
IBM Federal Systems Company Bethesda, MD
(301) 493-1496 locke@vnet.ibm.com

A Position Paper for the Workshop on Large, Distributed, Parallel Architecture Real-Time Systems Institute for Defense Analyses March, 1993

In recent years, the processes involved in the development of large, distributed, parallel real-time systems, responding to the major changes in the availability of high-performance hardware, have undergone a massive change due to the size and complexity of projects undertaken. Not every system developed in these environments has been completed successfully; it is clear that the difficulties associated with developing such systems have increased at least as fast as the increase in computing power. Thus, this workshop is being conducted in an environment in which many of the previous "rules of thumb" by which such systems have been conceptualized, procured, managed, and developed have become obsolete.

There are four questions posed by the Institute for Defense Analyses for consideration at this workshop. This position paper considers each question in turn.

INTRODUCTION

Question 1: What is the best method or methodology for designing large, distributed real-time systems where processing elements may have a parallel architecture?

There are several software design methodologies (e.g., Structured Analysis) which are frequently recommended and used for real-time system design, but none actually address the central requirements that distinguish a real-time system. The single characteristic of a real-time system which distinguishes it from other complex systems is its requirement to provide bounded response times (usually expressed in milliseconds elapsed between an input and its required external response) for all or part of its input domain. This is in sharp contrast to non-real-time systems for which the central performance requirement is generally stated in terms of throughput (usually expressed in rates, such as messages per second).

The requirement to provided bounded response time rather than some level of throughput creates a fundamental dichotomy between real-time systems and non-real-time systems which cannot be bridged by simply adding efficiency requirements to a design produced by any of the currently favored design methodologies. The requirement of bounded response time produces the need to manage all shared resources, including the processors, communications, memory, devices, and interfaces in ways that differ fundamentally from systems whose performance is characterized only by throughput requirements. For example, message queues for throughput-driven systems can generally use FIFO disciplines, while message queues for response time-driven systems must generally use time-driven queueing disciplines, such as priority or deadlines.

This fundamental difference must strongly affect the design process at the highest level. All basic system components -- operating system resource management, language run-time processes, device drivers, communications protocols at every level, and application software architecture -- must consistently reflect this fundamental difference. In distributed real-time systems being designed and built today and in the foreseeable future, most of these components are, and will increasingly be available as Commercial Off-The Shelf (COTS) components. The application software architecture must be constructed to use these components in such a way that its response requirements can be predictably and consistently met.

REAL-TIME SOFTWARE ARCHITECTURE

The application software architecture can be thought of as the highest level of application design. As with every level of design, it is expressible as a set of design decisions which will drive all other design decisions for the application. A real-time application software architecture, whether in a distributed, parallel or uniprocessor environment, which does not start with the consideration of its response time requirements, is highly likely to encounter serious performance problems later in its implementation life cycle, most frequently during integration and test. Design problems found during integration and test are among the most expensive to fix, particularly when they are traceable to the software architecture itself.

The software architecture defines just how the hardware will be used to satisfy the requirements. For each processing node, the architecture dictates which requirements will be supported, how communications will be handled (including contention and queueing disciplines), how concurrency and consistency will be provided and controlled, and how faults (either hardware, software, or both) will be detected, corrected, logged, and managed.

For real-time systems, the software architecture must start with the answers to several requirements-driven questions. Some of the most important of these questions are:

1. Is the design to be event driven or time driven? 2. How many different time constraints are present in the application? 3. Are the time constraints hard or soft? If soft, what is the nature of the actual constraints? If hard, are they primarily periodic, or aperiodic? 4. Are the inputs bounded in quantity and interarrival time? If not, what are their arrival characteristics? 5. Are the algorithms to be used characterized by bounded execution time?

The answers to these questions must drive the choice of software architecture, including the number of tasks (a task is a separately schedulable sequential procedure, without implications of presence or absence of shared information or communication with other tasks), use of priorities, communication techniques, and synchronization techniques. Only after these questions are answered should a design methodology be chosen. From the real-time perspective, the methodology choice can be arbitrary, but the requirements for bounded and predictable time management must be observed throughout the process, since the methodologies themselves will ignore that issue.

It is important to note that the presence of distribution and parallelization will have a significant effect on the choice of software architecture. At present, there are no completely general, well-understood architectures that can be easily analyzed to provide predictable response time in a distributed or parallel environment, in contrast to such techniques as rate-monotonic analysis or cyclic executives in uniprocessors. These simple uniprocessor techniques can be used for the individual processors in distributed parallel nodes, but analyzing the resulting system-wide response requires decomposing the external end-to-end time constraints into smaller individual constraints on partial computations, which is difficult and results in significantly sub-optimal resource utilization throughout the architecture.

A key paradigm to render system development cost manageable in today's large systems is reuse. With a few exceptions, the impact of software component reuse in the real-time systems under consideration in this workshop has been very low in the software architecture, design and implementation, but this must change. This required increase in the use of reusable components in these systems provides a further incentive to the creation of well-understood software architectures for large, distributed, parallel real-time systems, since there can be little utilization of reuse on the required scale unless the underlying software architectures are compatible.

SCHEDULING THEORY VERSUS DESIGN THEORY

Question 2: What should be the relationship between real-time Design Theory and real-time Scheduling Theory in a design methodology for this class of systems?

The fact that scheduling theory assumes knowledge of periodicity and execution times is not indicative of an implied assumption that the system has "already been designed", but is merely a reflection of the fundamental nature of real-time systems, in the same sense as the assumption in a missile design that the mass of the missile is known. In either case, even though the eventual design will certainly require changes and re-analysis throughout design and implementation, there is no substitute for early estimation and tracking of these parameters. There can be no assurance of the ability of a design to meet its timing constraints, either before or after the design is completed, regardless of the amount of testing, without this information. Note that the same is true of the cost of the system; obviously the cost cannot be precisely determined before the design and implementation is completed, but it will certainly be estimated as accurately as possible before any substantive technical effort is expended, and tracked (and updated) throughout the design and implementation of the system.

Additionally, the architecture that results from the answers to the architectural questions above will have a critical impact on the ability of the resulting design to meet its time requirements. Unlike many other design attributes, the ability to meet timing constraints is principally determined when the top-level architecture is defined, rather than in the myriad details that make up the remainder of the design. This is true because of the tight coupling between the scheduling (i.e., the sequencing of all system resources to meet time constraints) design and the successful performance of the system.

Thus, for the class of large, distributed, parallel real-time systems that are the subject of this workshop, it is critical that the Design Theory be heavily influenced by the Scheduling Theory. The present immaturity of both for such systems is strongly apparent, and is particularly evidenced by the fact that questions such as this are even being asked. The failure of a number of such systems that have been built over the last several years to meet their time constraints provides abundant evidence that it is the failure to create a real-time software architecture at the beginning of the design process that has led to designs incapable or barely capable of meeting time constraints.

As an example, consider the pipeline architecture which is frequently used for such systems. In this architecture, processing each input consists of a sequence of tasks, interconnected using any of several message passing techniques. Such an architecture seems intuitively simple due to the presence of multiple nodes in the distributed environment, and leads to significant amounts of concurrency and resulting efficiency. However, such designs do not meet the basic requirements of any of the known scheduling theories, all of which involve ensuring preference (e.g., priorities) to tasks with tighter time constraints, and explicit control over inversions of this preference, particularly in the presence of the COTS "open" systems components frequently proposed for them.

Thus, combining Design Theory and Scheduling Theory, for this class of systems, is mandatory, and should produce a Real-Time Architecture Theory which can then make extensive use of existing methodologies for subsequent decomposition into objects, abstract data types, or other popular constructions. The appropriate constructions for real-time systems can include only those that are analyzable using a sound Scheduling Theory if unrecognized real-time performance problems are to be avoided.

REAL-TIME VALIDATION AND VERIFICATION

Question 3: What is the best method for validating that large, distributed, parallel architecture real-time systems behave as specified?

The combination of Design and Scheduling Theories is particularly critical to the ability to validate the timing performance characteristics of a distributed real-time system for the same reason that Design Theory alone is critical in validating the correct functional behavior of large, complex systems. It is well known that software cannot be validated for correctness through testing alone; this is especially a problem when systems become very large. The problems of time correctness are, if anything, even more intractable, because a system with response time constraints will frequently behave properly a large part of the time, failing only intermittently. In fact, timing failures of real-time systems are almost always transitory (i.e., intermittent), with very similar characteristics as intermittent hardware.

This fundamental intractability in the ability to test a system for meeting its response time constraints renders it even more important to ensure that the software architecture conforms to a valid theoretical timing model that will make the analysis of the system tractable. Only in this way can the user be sure that not only does the system generally perform correctly, but that it is likely to continue to perform correctly, even in the presence of heavy loads (including overloads).

This does not mean, however, that there is no role for testing. The significance of testing relative to time constraints is to validate that the implemented system faithfully meets the individual time constraints of its components, leading to the assurance that the system as a whole will have the timing characteristics inherent in its design model. Similarly, the testing must verify that the system faithfully conforms to its architectural structure. Thus, for example, if the system architecture specifies that Task A sends messages only to Task B and Task C, that only those inter-task messages are transmitted by Task A, and that the message sizes fall within the limits defined.

DEVELOPING LARGE, DISTRIBUTED REAL-TIME SYSTEMS IN THE FUTURE

Question 4: Given that resources were available to enhance the design and testing methodologies for this class of systems, what are the most promising areas where these resources could be applied?

Clearly, developing large, distributed, parallel real-time systems successfully is an extremely complex process. The research domains most likely to yield important results would be those attacking the software architectural issues described previously in this paper. Important research has been ongoing for some years in the individual areas of real-time synchronization, communications, processor scheduling, memory management, and cache management, but additional work in combining results from these areas into coherent architectural strategies could help greatly. The disciplines required must combine scheduling, software engineering, and fault management at a minimum. The skills and experience base needed for such a research program would likely be available only through a team composed of both academic and development personnel.

November 9, 1992

Position Paper
on
Large, Distributed, Parallel Architecture, Real-Time Systems

Kjell Nielsen

Hughes Aircraft Company
PO Box 3310, MS: 618/B223, Fullerton, CA 92634

(714)732-3849 (office)
(714)732-1953 (fax)

This position paper addresses the four issues listed in the announcement for the Workshop on Large, Distributed, Parallel Architecture, Real-Time Systems to be held at IDA March 15-19, 1993.

The positions stated on the various issues include experiences gained and directions taken within Hughes regarding the system development process, design methodologies, and the use of CASE tools. Hughes is a large and diverse company and the stated positions do not necessarily reflect an overall, uniform company policy with respect to methodology and tools.

Most of the language dependent design and implementation issues are directed to Ada real-time systems, since Ada is the prevalent programming language used on a significant number of large, distributed, real-time systems being built. Ada also has a tasking model included at the application programming level.

In discussions addressing general concurrent elements, e.g., Unix or VMS processes, the term *process* is used. Concurrent elements in Ada are referred to as *tasks*.

1. What is the best method or methodology for designing large, distributed real-time systems where processing elements may have a parallel architecture?

It is not sufficient to only consider a design *methodology* for the development of large, distributed real-time systems. The scope should be expanded to include an integrated *system development process*. The process will employ several methodologies in the design and implementation of distributed systems. From an organizational point of view, systems engineers, hardware engineers, and software engineers should be involved in the process, i.e., *concurrent engineering.*

Embedded within this engineering process must be a set of consistent graphical representations that can be used to clearly identify products of individual steps in the process, and that will clearly show the transitioning activities from one step to the next. A system development process that embraces concurrent engineering should include the following steps:

1. Domain analysis.
2. System requirements analysis
3. System design
    a. Partitioning into
        - subsystems
        - hardware and software components
        - reusable software components

    b. Configuring (allocating requirements to hardware and software)
4. Hardware design
5. Software requirements analysis (for each partition)
6. Software design
    a. Process structuring (concurrency model)

    b. Language dependent design, e.g., for Ada
        - Ada task structuring
        - class/object structuring (Ada packaging)

    c. Software design evaluation

Two primary development approaches are currently in use for large, distributed systems. The *hardware-first* approach utilizes all of the steps listed above in the order suggested. There is a specific partitioning step to divide a large system into a set of suitable partitions, followed by (and iterated with) a configuring step which allocates system requirements to hardware and software entities (modules). The requirements represented within each module are designed and implemented in the specific programming language as virtual nodes (VNs) that are mapped to the hardware elements. In Ada, for example, VNs are collections of Ada packages, subprogram bodies, and task bodies that implement the set of requirements allocated to a hardware element during the partitioning/configuring activity. This is the traditional development approach for decomposing a large system into partitions that can be implemented as distributed processing elements (PEs).

The *software-first* approach skips the partitioning/configuring step of allocating requirements to hardware and software. The software requirements analysis is performed on a system wide basis (rather than for each partition), and software elements are developed before the hardware architecture has been determined. A hardware architecture is developed to support the software solution, and VNs are mapped to the hardware elements. This approach can be used successfully, for example, for the development of a product line, where small, medium, and large systems will be constructed within the same problem domain. Different hardware architectures are developed for the different size systems, and the software representing each system is composed of reusable software components.

A system development process and associated methodologies supporting the six steps outlined above include:

1. ART [HAC92] -- This is an integrated system development process that addresses all of the six steps outlined above. Domain analysis is included as the first activity to support reusability in-the-large [NIE92].
2. Real-Time Structured Analysis (RTSA) -- This is used as a methodology to support both the system requirements analysis, system design, and software requirements analysis. The methodology and notation is taken from Hatley-Pirbhai [HAT88], with supplemental information from Ward-Mellor [WAR85a, WAR85b, and MEL86], and Shumate-Keller [SHU92].

3. Information Modeling -- An information model is added to supplement the Hatley-Pirbhai process and control models. The notation for the Entity Relationship Diagrams (ERDs) is based on the description in [CHE76].
4. OOD with Ada -- This is an object-based Ada design methodology for large real-time systems [NIE88, NIE92, and SHU88a]. The transitioning from analysis to design is based on DARTS [GOM84]. Specific guidelines for creating a process abstraction and Ada task structuring is included [SHU88b]. The creation of VNs is based on certain structuring guidelines for distributed Ada programs [ATK88, JHA89, VOL89, NIE90].

The primary CASE tools used to support ART include Software Through Pictures (StP), developed by IDE, and Teamwork, developed by Cadre. Neither of these two tools is fully integrated to support the complete development process. Both vendors are currently working to improve the tool support for additional steps of the development process.

It should be noted that none of the development steps or tools mentioned support hardware design.

2. What should be the relationship between real-time Design Theory and real-time Scheduling Theory in a design methodology for this class of systems?

Real-time design theory refers to a collection of features that pertain to a concurrency model of multiple processes executing in parallel on distributed PEs. Processes may also compete for the use of the same PE, i.e., *apparent concurrency*, as opposed to processes executing in different PEs with *real concurrency*. We need to be able to handle both conditions. Some of the features of real-time design theory [BEN82, LEV90] that apply to scheduling include:

- Safety -- a concurrent element (process) must perform correctly independent of the other processes in the system.
- Liveness -- two or more processes must exhibit the correct behavior in the dynamic environment of asynchronous execution.

- Adequate response time for critical events -- e.g., interrupts must be serviced in a timely fashion.
- Schedulability -- processes must be scheduled to execute and complete their functions under certain system dependent time constraints.
- Overload response -- if all the processes in the system cannot meet their deadlines, the selected critical processes must still be serviced to meet their deadlines.
- Mutual exclusion -- certain sequences of instructions will be expected to execute within a critical section. This does not apply directly to scheduling events, but applies to distributed databases and multiple access of shared data, and is important during synchronization of processes.
- Priority inheritance -- the precise timing of a priority assigned to a process that will execute next during synchronization of two processes with different priorities (e.g., in Ada the priority of the highest task is assigned at the start of the rendezvous).

Real-time scheduling theory forms the basis for implementing the scheduling features of the real-time design theory. It also allows the implementers of real-time systems to analyze timing correctness and make predictions about the expected system performance. Elements of real-time scheduling theory [SHA90, and LEV90] include:

- Scheduling mechanism -- can be based on round-robin, time-slicing (deterministic) or a "fair" selection method with preemption (non-deterministic).
- Scheduling algorithms -- used to predict whether or not time-critical processes will complete execution within required timing restrictions. The most prevalent of these is hard deadline scheduling based on a set of periodic processes: Rate-monotonic algorithm where the processes are assigned priorities in reverse relation to their periodicities (shortest period given highest priority).
- Fairness in scheduling -- can be implemented by dynamically increasing the priority of a process as the criticality increases (e.g., making a controlled object avoid hitting an obstacle).

The current design methodologies should be expanded to include guidelines for dealing with real-time considerations (e.g., safety, liveness, schedulability, etc.). Specific heuristics should be developed for handling the scheduling of concurrent processing elements. This should include aperiodic as well as periodic tasks. These expanded design guidelines should be based on known scheduling theories that can reasonably be expected to be available in run-time models, e.g., the current preemptive Ada tasking model and proposed rate-monotonic mechanisms.

The design theory we use in our design methodology must be implemented in the semantics of the process abstraction model. For example, it does not make sense to try to implement critical, periodic tasks in the current version of Ada-1983 which has a run-time implementation of an "at least" condition for the expiration of a periodic task. It is important that design theory methodologists and run-time implementers communicate effectively about their respective needs and possible run-time implementation problems. As future improvements of run-time systems are developed for real-time systems, we must avoid the frustrating situations of the early Ada systems when the designers tried to implement designs based on preemptive scheduling that did not exist in the implementation model. The run-time developers had interpreted the Ada Reference Manual to mean that preemption was not required.

3. What is the best method for validating that large, distributed, parallel architecture real-time systems behave as specified?

The validation of large, distributed real-time systems includes three primary elements: (1) a set of plans and procedures for how the validation is to be performed; (2) a training plan to ensure that the developers are implementing the test plans and procedures in a consistent manner; and (3) a set of modeling and test tools to support the validation process.

DoD-Std-2167A has received considerable (justified) criticism with regard to a literal interpretation of the contents and order of its numerous analysis and design documents, and the implication that it tends to dictate a design methodology. Such a literal interpretation should, however, be encouraged for the 2167A set of test documents which include a Software Test

Plan, Software Test Description, and Software Test Report. An early focus of the contents of these documents (even for commercial projects where 2167A is not required) forces attention to the test phase as a process. A description of test cases is prepared before the actual test phase begins, and helps to identify the efforts required for unit testing and integration testing. Particular attention should be paid to test cases and validation procedures to analyze the system for deadlock, starvation, data integrity, schedulability of processes, and communication performance.

A significant amount of training may be required before the test phase begins to ensure that the developers understand the kind of testing required, and that they will be following the test procedures. This is even more important for distributed real-time systems where considerable challenges are presented for validating real-time performance requirements. In many cases we are finding that the developers are merely *trying to get a system to run during the test phase*, when they should, instead, be *testing a running system.*

There is, unfortunately, no unique list of support tools that will guarantee the complete validation of a distributed real-time system. Useful test tools include code analyzers, static analyzers, test probes, and modeling tools. Formal methods embedded within these tools must be clearly understood, in particular, with regard to their limitations. Results in the form of metrics must be used sensibly, and do not represent "proof of correctness." The use of automated test tools should be encouraged during the entire development period to

The overall test philosophy should be based on finding bugs as early as possible in the development cycle. The validation process should occur throughout the development cycle to give us a better product delivered to the customer. To support this philosophy, we need a consistent error reporting mechanism throughout the development cycle. A concentration of design and coding errors in certain functional areas will focus our testing efforts to those areas (but not to the detriment of other areas).

4. Given that resources were available to enhance the design and testing methodologies for this class of systems, what are the most promising areas where these resources could be applied?

4.1 Design Methodologies and Tools

The primary key to reusable designs in distributed systems is the degree of transparency of the inter-process and inter-processor communication (IPC) mechanism. Most of the distributed systems implemented today are designed with a unique IPC mechanism of the "not-invented-here" variety. This is also true to some extent for the underlying communication protocols regarding the number of layers that are implemented.

Specific design guidelines should be established for the creation of standardized IPC mechanisms based on the required functionality, e.g., broadcast, synchronous and asynchronous connection-oriented communication, multicast, etc. The guidelines should include the use of message passing, remote procedure calls (RPC), remote entry calls (REC), and the use of shared data in heterogeneous and homogeneous architectures.

A set of standardized interfaces (bindings) should be developed for Ada, C, and C++ programs for each of the IPC mechanisms developed. This will promote truly reusable programs at the application interface level.

Design guidelines should be developed for implementing Ada, C, C++, and mixed language programs in distributed architectures. This should include alternatives to the use of the Ada tasking model.

Design guidelines should be developed for distributed database design including a set of standardized locking mechanisms.

Funds should be made available to support the major tool vendors for improving existing CASE tools for the most promising system design methodologies.

4.2 Testing Methodologies and Tools

An important element of validating large, distributed real-time systems is the use of prototyping. The traditional method of prototyping includes the use of throw-away code as the complete system is implemented. A better approach is to develop a set of prototyping tools that can aid in the debugging and understanding of the system to be implemented, without developing throw-away code. An example of such a tool is a device to simulate a particular bus or LAN interface, e.g., Mil-Std-1553 or Ethernet. This device will simulate the bus or LAN functions (e.g., the arbitration mechanism) and record the stimulus/response activity with the distributed system. Such a device can be an invaluable aid in understanding the distributed system in terms of transient functions like startup, restart, and error detection and recovery.

A set of test cases has been developed for measuring the performance of real-time features in Ada programs in uniprocessor architectures (available from SIGAda's PIWG). A similar set of test cases could be developed for measuring the performance of programs in multiprocessor architectures. Particular performance features could include communication time latency and the efficiency of the IPC mechanism., and schedulability. Special code analyzers could be developed to predict the potential for deadlock and starvation. The reduction of these real-time risk areas before testing starts would greatly enhance the validation process.

Dynamic analyzers can be developed to measure the performance of a distributed system in a truly asynchronous environment. The currently available static analyzers don't help us here.

References

ATK88 Atkinson, C. et al., *Ada for Distributed Systems*, Cambridge University Press, Cambridge, England ,1988.

BEN82 Ben-Ari, M., *Principles of Concurrent Programming*, Prentice-Hall International, Inc., Englewood Cliffs, NJ, 1982.

CHE76 Chen, P., The Entity-Relationship Model — Toward a Unified View of Data, *ACM Trans. on Database Systems*, Volume 1, Number 1, 1976.

GOM84 Gomaa, H., A Software Design Method for Real-Time Systems, *Comm. ACM*, Volume 27, Number 9, September, 1984.

HAC92 *ART Guidebook -- Volume I: Development Process and Methodology; Volume II: Case Studies and Exercises*, Hughes Aircraft Company, October 1992.

HAT88 Hatley, D.J. and Pirbhai, I.A., *Strategies for Real-Time System Specification*, Dorset House Publishing, New York, 1988.

JHA89 Jha, R. et al., Ada Program Partitioning Language: A Notation for Distributing Ada Programs, *IEEE Transactions on Software Engineering*, Volume 15, Number 3, March 1989.

LEV90 Levi, S-T. and Agrawala, A.K., *Real-Time System Design*, McGraw-Hill, New York, 1990.

MEL86 Mellor, S.J. and Ward, P.T., *Structured Development for Real-Time Systems, Volume 3: Implementation Modeling Techniques*, Yourdon Press, New York, NY, 1986.

NIE88 Nielsen, K.W. and Shumate, K., *Designing Large Real-Time Systems with Ada*, McGraw-Hill, New York, 1988.

NIE90 Nielsen, K.W., *Ada in Distributed Real-Time Systems*, McGraw-Hill, New York, 1990.

NIE92 Nielsen, K.W., *Object-Oriented Design with Ada: Maximizing Reusability for Real-Time Systems*, Bantam Books, New York, 1992.

SHA90 Sha, L. and Goodenough, J.B., Real-Time Scheduling Theory and Ada, *Computer*, July 1987.

SHU88a Shumate, K., Layered Virtual Machines/Object-Oriented Design (LVM/OOD), in *Proceedings of the Fifth Washington Ada Symposium*, ACM, June 27-30, 1988, Washington, DC.

SHU88b Shumate, K., *Understanding Concurrency in Ada*, McGraw-Hill, New York, 1988.

SHU92 Shumate, K. and Keller, M., *Software Specification and Design: A Disciplined Approach for Real-Time Systems*, John Wiley, New York, 1992.

VOL89 Volz, R.A. et al., Translation and Execution of Distributed Ada Programs: Is It Still Ada? *IEEE Transactions on Software Engineering*, Volume 15, Number 3, March 1989.

WAR85a Ward, P.T. and Mellor, S.J., *Structured Development for Real-Time Systems, Volume 1: Introduction & Tools*, Yourdon Press, New York, NY, 1985.

WAR85b Ward, P.T. and Mellor, S.J., *Structured Development for Real-Time Systems, Volume 2: Essential Modeling Techniques*, Yourdon Press, New York, NY, 1985.

Large-Scale Distributed Real-Time Computing

Lui Sha
Ragunathan Rajkumar
Software Engineering Institute
Carnegie Mellon University

1.0 Introduction

Real-time computing and communication systems are critical to an industrialized nation's technological infrastructure. Modern telecommunication systems, automated factories, defense systems and air-traffic control systems cannot operate without them. Indeed, real-time computing and communication systems control the very systems that keep us productive, make our manufacturing processes competitive, enhance our security, and enable us to explore new frontiers of science and engineering. The explosive growth of applications executed dependably in real time is envisioned in diverse areas such as battlefield simulations, C^3I systems, high-bandwidth multimedia communications, and distributed flexible manufacturing.

The key requirements for advanced real-time systems are *predictability*, *dependability* and *performance*. A real-time system's timing behavior should be predictable before it is developed or modified. The system must have the ability to tolerate the failure of individual subsystems and provide a high degree of performance. The most significant developments in these areas are the generalized rate-monotonic scheduling theory that provides a theoretical foundation for the development of predictable real-time systems, the membership-based fault-tolerance protocols that allow flexible management of redundant resources, gigabit networking technology, high-performance RISC processors and parallel processing architectures.

The DoD 1991 *Software Technology Strategy* document refers to RMS as a "major payoff" and states that "System designers can use this theory to predict whether task deadlines will be met long before the costly implementation phase of a project begins. It also eases the process of making modifications to application software,..." The Acting Deputy Administrator of NASA recently stated in a 1992 speech entitled *Charting The Future*, "Through the development of Rate Monotonic Scheduling, we now have a system that will allow (Space Station) Freedom's computers to budget their time, to choose between a variety of tasks, and decide not only which one to do first but how much time to spend in the process." The RMS approach is also cited in the Selected Accomplishments section of the National Research Council's 1992 report, *A Broader Agenda for Computer Science and Engineering*. Our GRMS approach has also been rapidly gaining acceptance in the industry, and has been applied to national high-technology projects such as BSY-1, BSY-2 and NASA's Space Station. Scheduling support for the use of generalized RMS can now be found in major national hardware and software standards such as Ada 9x, IEEE POSIX.4, and the IEEE Futurebus+ bus standard.

To advance the state of real-time computing, it is important to build upon these successes. Thus, we propose to extend GRMS [2,3,4,7] in the context of a very large-scale distributed computing system where the communication delays make it impossible for each scheduler to have timely and complete system state information. Furthermore, we must create a unified framework for high-performance real-time fault-tolerant computing. This unified framework should provide an application infrastructure that employs high-performance computers and networks. Such systems must handle high-volume synchronized video, audio and text as well as real-time data streams with different periodicities and latency requirements from radar, hydrophones, satellite and other measurement instruments. Users of these systems can virtually visit different geographical locations, and get a "first-hand view" of the situation. Global assessment can be facilitated by gathering information from different points at a single decision point. In large-scale complex systems, component failures and application software errors are inevitable. The ability of the unified framework to deal with software and hardware errors can greatly enhance the reliability, functionality and flexibility of C^3I systems, air traffic control systems, modern mass-transportation systems, automated factories and defense applications such as nationwide battlefield simulations.

2.0 Fundamental Challenges

To develop an application infrastructure for large-scale distributed real-time computing, there are some fundamental challenges that we must meet.

- Decisions in this distributed environment must be made in decentralized fashion from both dependability and performance points of view. However, due to the geographical distribution of subsystems, propagation delays can be excessive, and decisions must be based on delayed and sometimes even incomplete information. Nevertheless, the distributed scheduling actions must be consistent.
- Dependability requires that individual subsystem faults do not crash the entire system. In particular, two critical yet complementary aspects must be addressed. First, an approach to deal with the increasingly serious problem of application software errors is necessary. Secondly, we need the analytical foundations and system primitives to deal with failures of hardware and software resources.

2.1 Maintaining Coherence in Distributed Scheduling Actions

As network speed and the physical distances between nodes increase, the state of the system is distributed. This has been recognized as a key problem of the *Core CS&E research Agenda for the Future* of the 1992 National Research Council's report titled, *A Broader Agenda for Computer Science and Engineering*. It states, "A network is an interconnected system, with many possible paths for feedback to any given node......the inability to predict just when these feedback effects will occur presents many problems for system designers concerned about avoiding catastrophic positive feedback loops that can rapidly consume all available bandwidth".

Fortunately, we have already solved this problem for the special case of a wide area dual-link network [5]. A dual-link network consists of two unidirectional links carrying traffic in fixed size cells in opposite directions. This can be considered as a special case of a network of switches with only two connections. The bandwidth usage requirements of downstream stations is fed back by setting a request bit in a cell flowing in the opposite direction. However, such feedback is delayed due to large distance. Furthermore the bandwidth requirements of upstream stations will never be known by the downstream stations. Thus, stations in a large high speed network must make scheduling decisions with incomplete and delayed information. The challenge is to achieve predictable operation under these circumstances.

We have developed a theory of coherent dual-link networks and a coherent scheduling protocol that ensures that the system will be consistent despite its distributed state [5]. Under this protocol, traffic in a dual-link network is transmission-schedulable[1] when an equivalent centralized system is schedulable. It is important to generalize this notion to an arbitrary network topology.

2.2 Software Fault -Tolerance and Analytic Redundancy

Statistics of large computing systems show that the probability of a system failure due to software bugs is about ten times that due to hardware faults. Therefore, in brief, we must be able to deal with software errors to have a reliable real-time system. To deal with software faults, some form of redundancy in computation is needed. Direct redundancy uses different programs to compute the same results so that voting or mid-value selection can be used. An example of direct redundancy for software fault tolerance is N-version programming. The fundamental problem with direct redundancy is that it is costly and independently developed software can still have common errors. The source of software faults is complexity.To successfully deal with software faults, we must let simplicity control complexity.

This approach is realized by the use of analytic redundancy [6]. Under this approach, programs with different complexities will compute different results that are analytically related. Particularly, we will develop a trusted simple software system which will give us a baseline answer on time plus a set of confidence assertions. A confidence assertion is a generalization of the statistical concept of confidence interval, which creates an "envelope" within which the solutions from the complex must lie. The complex software is not trusted. Its outputs must be consistent with the confidence assertions produced by the simple software or they will be discarded. Furthermore, we are not even able to trust the computation process employed by the complex software, which may have serious bugs that can crash the computation process itself.

To illustrate the use of analytic redundancy, we found that it was useful to classify software errors into three types for the purpose of detection and recovery. We shall use tracking as an example to illustrate this concept.

- *Inaccuracy*: In the context of our applications, these are tracking errors, which are a function of the quality of the data and the sophistication of the tracking algorithm. Due to the nature of the application, such errors can only be reduced but not eliminated. Design or implementation errors in software development can also contribute to tracking errors.
- *Timing faults*: These typically occur in the form of missed deadlines. While software design and implementation errors may lead to timing faults, a major source of timing faults is the time-complexity of the algorithms. The difficulty in tracking applications is that sophisticated algorithms may reduce the number of tracking errors but contribute to timing faults.
- *Programming system faults*: These are those serious software faults that may crash the system, for example, illegal addresses or data, exhausting available buffers, monopolizing the I/O channels and/or CPU.

Figure 1 is a model which compares the characteristics of a simple software system with those of a complex software system.

The software architecture used to deal with these faults is known as the *Simplex Software Architecture*. This architecture employs: (1) analytic redundancy to guard against application level software errors, (2) runtime isolation and fault containment techniques to guard against programming system level software errors such as illegal addressing, and (3) generalized rate -monotonic

[1] A connection is transmission-schedulable if it can guarantee transmission of C cells per period T.

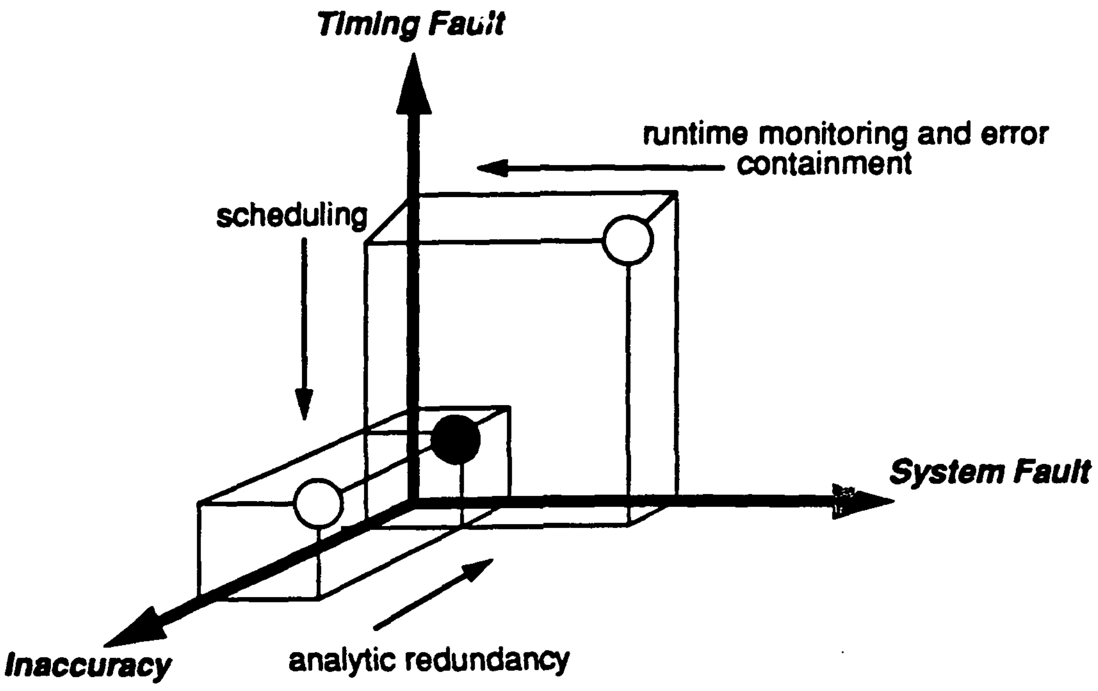


FIGURE 1. Conceptual Model to build Dependable Real-Time Systems

scheduling techniques to guard against timing errors. Figure 1 is the conceptual model that illustrates the combined use of scheduling, runtime fault containment and analytic redundancy to improve the overall functional performance and reliability.

2.3 System-Level Failure Management in Distributed Real-Time Systems

Distributed gigabit network based real-time systems must be robust and fault-tolerant. However, the construction of distributed fault-tolerant real-time systems bring new challenges. Processor and network scheduling must be carried out coherently with the support mechanisms for fault-tolerance. In addition, these integrated mechanisms must be supported by all system layers. Generalized RMS already provides a solid foundation in processor and bus scheduling for real-time systems. A critical system issue is the need for application-independent support at the system-level to build dependable real-time systems. Such support will greatly enhance the ability to tolerate a wide range of system faults (along the system fault dimension of Figure 1) including the failures of processors, communication links and interfaces, process creation, and communication protocols.

Traditionally, there has been a misconception that priority-based scheduling techniques cannot ensure determinism when redundancy techniques are used. As a result, most if not all real-time fault-tolerant systems use cyclical executives that employ lock-step execution and comparison of redundant components. However, the only necessary correctness criterion is the need to maintain I/O determinism in redundancy management and the interface to the external environment.

GRMS can be used as the basis to provide I/O determinism while still allowing different programs to execute on redundant processors. However, the protocols to detect and recover from faults on a timely basis must be developed. The critical factor for developing these protocols is that a timely and consistent view of the state of the distributed system resources is maintained despite failures.

We are currently developing an analytical approach and system primitives to provide system-level support for tolerating and/or recovering from processor, process and communication failures in distributed real-time systems. The key element behind system-level fault-tolerance is the real-time management of spatial redundancy to achieve dependable system operation even in the presence of resource failures.

3.0 Summary and Conclusion

Real-time computing and communication systems are critical to an industrialized nation's technological infrastructure. Modem telecommunication systems, automated factories, defense systems and air-traffic control systems cannot operate without them. The key requirements for advanced large-scale real-time systems are *predictability, dependability* and *performance*. A real-time system's timing behavior should be predictable before it is developed or modified. It should have the ability to tolerate the failure of individual subsystems while providing a high degree of performance. Significant developments in these areas are the generalized rate-monotonic scheduling theory which provides a theoretical foundation for developing predictable real-time systems, membership-based fault-tolerance protocols for flexible management of redundant resources, wide-area gigabit networks, and high-performance RISC and parallel processing architectures. , It is important to build upon the successes of GRMS in industry, high-technology projects and commercial standards.

We propose to extend GRMS in the context of a very large-scale distributed computing system where the communication delays make it impossible for each scheduler to have timely and complete system state information. Furthermore, we must create a unified framework for high performance real-time fault tolerant computing. This unified framework should provide an application infrastructure that allows us to develop advanced large-scalereal-time systems.

References

[1] J. P. Lehoczky and L. Sha, "Performance of real-time bus scheduling algorithms," ACM Performance Evaluation Review, Special Issue, vol. 14, May 1986.

[2] C. Liu and J. Layland, "Scheduling algorithms for multiprogramming in a hard real-time environment," Journal of the ACM, vol. 30, pp. 46-61, January 1973.

[3] L. Sha and J. B. Goodenough, "Real-time scheduling theory and Ada," IEEE Computer, vol. 23, pp. 53-62, April 1990.

[4] L. Sha, R. Rajkumar, and J. P. Lehoczky, "Priority inheritance protocols: An approach to real-time synchronization," IEEE Transactions on Computers, vol. 39, pp. 1175-1185, September 1990.

[5] L. Sha, S. Sathaye, and J. K. Strosnider, "Scheduling real-time communication on dual link networks," 13th IEEE Real-Time Systems Symposium, December 1992.

[6] L. Sha, J. P. Lehoczky, M. Bodson, P. Krupp and C. Nowacki, "Responsive Airborne Radar Systems", Proceedings of the 2nd International Workshop on Responsive Computer Systems, October, 1992.

[7] H. M. B. Sprunt, L. Sha and J. P. Lehoczky, "Aperiodic Task Scheduling for Hard Real-Time Systems", Real-Time Systems Journal, 1989.

An Integrated Approach to Design and Development of Large Distributed Real-Time Systems

A Position Statement

Sang H. Son
Department of Computer Science
University of Virginia
Charlottesville, Virginia 22903

As the complexity of new applications of large distributed real-time systems increases, so does the need for improvement for real-time system design and development methodology. The critical nature of many real-time systems requires a rigorous design and development of their components, and validation of timing characteristics. The traditional approach that carries out the tasks of system modeling, timing verification, and system implementation rather independently, seems inadequate for developing a large distributed real-time system partly because

(1) verification of timing-related properties has limitations, especially in distributed/parallel environments,

(2) timing characteristics are very hard to determine in the early stages of design,

(3) it may introduce inconsistencies between the model and the implemented system,

(4) due to high cost and long development time, it is often too late when problems are discovered at system integration time.

For example, it is very hard to determine, during the design phase, the synchronization and communication requirements among tasks that will be distributed to several nodes in the implemented system. However, the scheduler at each node should rely on that information to provide predictable timing

This work was supported in part by ONR by CIT, and by IBM Federal Sectors Division.

behavior. Furthermore, to have a complete design, we need to decide which scheduling algorithms are to be used for which resources at design time. We can make worst case assumptions in many cases, but there should be a facility to test the impact of the design assumptions on the timing characteristics of the system. Making worst case assumptions for designing a real-time system that can ensure correct operation even in the situations with maximum need potentially wastes large amounts of resources. In some applications of large distributed real-time systems, the worst case need may be unbounded.

It seems clear that we need a new integrated approach to design, development, and verification of large distributed real-time systems. It should provide a facility to evaluate the timing constraints in the early stage of the system design, and to monitor their impact during system implementation. Such an environment dedicated to the design and development of real-time systems must support many facilities that are not present in current programming environments. Recently, there have been attempts to provide an integrated environment for real-time system design, development, and evaluation [Son92, Bar91, Ran91, Jah91] However, this is the area to which more support for active research and investigation seems necessary due to its high potential for significant benefits to overall system design and development.

Given the functional and timing specifications for a real-time system, the first challenge is to validate that there exists no inconsistency in the specification. This is a non-trivial task. Even though there has been a considerable research effort in this area of verifying the specification (e.g., RTL and Modechart from UT Austin [Stu90]), applicability of those formal methods to practical problems has several limitations.

Assuming that the specification is validated to be consistent and feasible, we need to come up with an initial system design. At this stage, we only have very rough idea about resource and synchronization requirements of tasks. The integrated environment should provide a tool that can help the designer to develop a top-level design from the given specification. The object-oriented approach seems appropriate for this type of tool, because the external behavior of each component (or object) can be specified without going through the internal implementation details. Even though there are several tools and methods developed for real-time system design (e.g., [Fau92]), their capabilities are not tested yet for large and complex real-time systems.

The next step is to develop a prototype of the system according to the initial design. The prototype consists of modules that represent system components in the initial design. Modules for which the implementation has not been determined or for the hardware component which is not yet available can be simulated. The simulated part estimates resource/synchronization requirements of the physical object that it represents. The timing constraints and functionalities of the given specification can be tested using the prototype. If the initial design does not satisfy the given specification, the design should be refined. In some cases, the initial design may need to be abandoned and totally redesigned. This refinement process will continue until we have a stablized system design and prototype that satisfies all the requirements. By following this iterative refinement and its prototyping, we can evaluate the impacts of the design choice early in the design stage and make necessary changes.

One of the benefits of this integrated design approach is that the designer can check out whether the design philosophy under which the system is being developed is appropriate for the current application. For example, in the early design stage, we need to decide on the philosophy for resource management. In most real-time systems, the responsibility of resource management is typically shared by the operating system and the application, partly because it is the application that knows about requirements and semantic information necessary to support timeliness even in the presence of overloads and faults. There is a spectrum of design approaches to dividing responsibilities between the two, and the decision depends primarily on the design philosophy and methods used to build applications [Nat92]. At one extreme, the cperating system provides no special support, and the total responsibility is on the application. At the other extreme, the operating system takes all the responsibility for scheduling with no information from the application. These two extremes are convenient in the sense that the operating system and the application do not need to share application-specific semantics. However, for the same reason, the capabilities of those approaches are inherently limited. Using the integrated approach, we can test out not only two extremes, but also different approaches rather easily.

Other example to demonstrate the benefits of this approach is the choice of scheduling algorithms/policies. Contrasted with non-real-time systems in which a relatively simple scheduler chooses a ready job non-deterministically without considering timing requirements, schedulers in real-time systems must use a variety of information and selection criteria. The many choices and variations in terms of scheduling policies makes it almost impossible to know which choice would perform better, without actually testing them against the given requirements. If we know the execution time and blocking time of each task, we may be able to perform schedulability analysis using certain scheduling theories. However, those timing characteristics can be estimated only after we determine the scheduling policies. This shows why the integrated design approach combined with prototyping is beneficial. We can plug in different scheduling policies into the prototype and test their timing behavior.

Another important requirement for the integrated approach is to provide modeling capability for not only the target system but also the operating environment. To achieve that, the integrated design approach should support the running of the prototype under the proposed operating environment. Some of the facilities that are necessary include

(1) generate external events,

(2) change the values of conditions,

(3) update variables and other data elements,

(4) trigger state changes,

(5) activate/deactivate task activities.

The testing/debugging phase usually constitutes a large proportion of the total system development time. Due to the critical role played by large distributed real-time systems, it is almost always necessary to enforce the highest level of quality assurance to be employed for testing. With the advent of more ambitious applications of large distributed real-time systems, such as NASA's space station project, testing and validating the quality of the developed software becomes more costly and time consuming. Any small reduction of the complexity of the testing phase, while maintaining the same guarantee of system performance, may result in a substantial benefit to the system development effort. The integrated design and development approach can substantially reduce the amount of work involved in testing to ensure timing constraints.

To summarize, the major advantages of the integrated approach to design and development of real-time systems include

(1) It allows timing properties of a real-time system being designed/developed to be analyzed in early stages of the system development cycle.

(2) It allows the functional correctness to be tested, while permitting reduced effort to redesign the system and/or components.

(3) It allows to verify the assumptions made during the design phase.

(4) It encourages reusability of system components.

References

[Bar91] R. Bargodia and C. Shen, MIDAS: Integrated Design and Simulation of Distributed Systems, *IEEE Trans. on Software Engr.*, vol. 17, no. 10, Oct. 1991.

[Fau92] S. Faulk et al, The Core Method for Real-Time Requirements, *IEEE Software*, vol. 9, no. 5, Sept. 1992.

[Jah91] F. Jahanian and R. Rajkumar, An Integrated Approach to Monitoring and Scheduling in Real-Time Systems, *IEEE Workshop on Real-Time Operating Systems and Software*, Atlanta, Georgia, May 1991.

[Nat92] S. Natarajan and W. Zhao, Issues in Building Dynamic Real-Time Systems, *IEEE Software*, vol. 9, no. 5, Sept. 1992.

[Ran91] K. Ransom, C. Marlin, and W. Zhao, An Integrated Environment for the development and Analysis of Hard Real-Time Systems, *IEEE Workshop on Real-Time Operating Systems and Software*, Atlanta, Georgia, May 1991.

[Son92] S. H. Son, An Environment for Integrated Development and Evaluation of Real-Time Distributed Database Systems, *Journal of Systems Integration*, vol. 2, no. 1, February 1992.

[Stu90] D. Stuart, Implementing a Verifier for Real-Time Systems, RTSS '90, December 1990.

Guarantees of Hard Real-Time Communications in FDDI Networks[1]

Wei Zhao
Department of Computer Science
Texas A&M University
College Station, TX 77843
zhao@cs.tamu.edu

1 Introduction

We concern with deadline guarantees in distributed hard real-time systems. In particular, we address issues in guaranteeing hard real-time message delivery in an FDDI (Fiber Distributed Data Interface) network.

It has become a common practice to use digital computers for embedded real-time applications such as space vehicle systems, image processing and transmission, and integration of expert systems into avionics and industrial process control. A salient feature of these computations is that they have stringent timing requirements. A timing failure could lead to catastrophe. Further, these systems are often distributed. This is not only because the applications themselves are often physically distributed, but also due to the potential that distributed systems have for providing good reliability, good resource sharing, and good extensibility [19, 44, 52].

The key to success in using a distributed system for these applications is the timely execution of computation tasks that usually reside on different nodes and communicate with one another to accomplish a common goal. End-to-end deadline guarantees are not possible without a communication network that supports the timely delivery of inter-task messages. On the other hand, despite efforts to make the system reliable, faults may still occur due to a severe working environment and failing components. The main focus of our work is to address some important issues related to fault-tolerant guarantees of synchronous message deadlines, i.e., no matter what happens (even in the presence of a network fault), the messages will be transmitted before their deadlines.

We have selected FDDI (Fiber Distributed Data Interface) networks for this study. FDDI is an ANSI standard for a 100 Mbits/sec fiber optic token ring network [2, 3]. FDDI is suitable for real-time application not only because of its high bandwidth but also due to its bounded access time and its dual ring architecture. Since the early 1980's, extensive research has been done on the FDDI networks. The FDDI MAC protocol was first proposed by Grow [12]. Ross [34, 35, 36], Iyer and Joshi [14, 15] and others [25, 43] provided comprehensive discussions on the timed token protocol and its use in the FDDI. Many new civil and military networks are being developed based on the skeleton of FDDI. Examples include the High-Speed Data Bus and the High-Speed Ring Bus (HSDB/HSRB) [37, 38, 46], the Survivable Adaptable Fiber Optic Embedded Network (SAFENET) [11, 20, 26], and FDDN (Fiber Distributed Data Network) [9]. Many embedded real-time applications use FDDI as backbone networks. For example, FDDI has been selected as a backbone network for NASA's Space Station Freedom [8, 7, 50].

Our work is motivated by recent advances in the theory of hard real-time scheduling[48, 49]. For real-time systems, the basic design requrements for a communication protocol and for a centralized scheduling algorithm are similar: both are constrained by time to allocate a serially used resource to a set of processes. Liu and Layland [23] addressed the issue of guaranteeing the deadlines of synchronous (i.e., periodic) tasks in a single CPU environment. They analyzed a fixed priority preemptive algorithm, called the *rate monotonic algorithm*, that assigns priorities in inverse relation to task's periods. They showed that the *Worst Case Achievable Utilization* of the algorithm is 69%. Provided that the the utilization of the task set is no more than 69%, the task deadlines are always guaranteed to be satisfied. The algorithm was also proven to be optimal among all fixed priority scheduling algorithms in terms of achieving the highest worst case utilization. The rate monotonic

[1] This work is supported in part by grants from the Air Force Office of Scientific Research, the National Science Foundation, the Office of Naval Research, the Research Institute for Computing and Information Systems of the University of Houston - Clear Lake, and Texas A&M University.

scheduling algorithm has been subsequently extended by many researchers [40], and is used in many hard real-time applications [10].

Intuitively, one would believe that a communication protocol that implements the rate monotonic transmission policy is the most desirable for a real-time communication environment. However, implementation of the rate monotonic policy requires global priority arbitration every time a node in the network is ready to transmit a new message. FDDI does not support priority arbitration at the medium access control level. Consequently, it is difficult, if not impossible, to implement the rate monotonic transmission policy in an FDDI network.

However, the methodology for analyzing the rate monotonic algorithm has a more profound significance than merely its relevance to the fixed priority preemptive algorithms. The methodology stresses the fundamental requirements of *predictability* and of *stability* in hard real-time environments and is therefore also befitting to other hard real-time scheduling problems. In this methodology the Worst Case Achievable Utilization is used as a metric for evaluating the predictability of a scheduling algorithm. As long as the CPU utilization of all tasks is within the bounds specified by the metric, all tasks will meet their deadlines. This metric also gives a measure of the stability of the scheduling algorithm in the sense that the tasks can be freely modified as long as their total utilization is held within the limit. Because of this, we adopted the same methodology in our study of guaranteeing message deadlines in FDDI networks. We analyze the run-time control schemes of FDDI networks for hard real-time communication based on the Worst Case Achievable Utilization.

2 Network and Message Models

We consider a network consisting of two counter-rotating rings. Each ring consists of m nodes connected by point-to-point links forming a circle i.e., the token ring. The two rings will be denoted *ring A* and *ring B*. We denote the ring latency by τ which includes the ring propagation delay, the node latency delay, the transmission delay of the token, etc. Thus, τ is the walk time of the token when none of the nodes disturb it. The ratio of the ring latency τ to the target token rotation time (TTRT) is denoted by α. The usable ring utilization would therefore be $(1 - \alpha)$ [47].

A node can connect either to one of the rings or to both. A node can transmit and receive messages from a ring only if the node connects to it. For those nodes that are connected to two rings, we assume that they have dual facilities for transmitting and receiving messages on both rings. Hence, the node can simultaneously transmit/receive messages on both rings.

Messages generated in the system at run time may be classified as either *synchronous messages* or *asynchronous messages*. We assume that there are n_A (n_B) streams of synchronous messages, $S_1, S_2, \ldots, S_{n_A}(S_{n_B})$ in the system which form a synchronous *message set, $M_A(M_B)$,* for ring A (ring B), i.e.,

$$M_A = \{S_1, S_2, \ldots, S_{n_A}\} \tag{1}$$

and

$$M_B = \{S_1, S_2, \ldots, S_{n_B}\}. \tag{2}$$

For the convenience of our discussion, we use the notation M to denote either M_A or M_B. Similarly, n denotes either n_A or n_B.

Messages have the following characteristics:

1. Synchronous messages are *periodic*, i.e., messages in a synchronous message stream have a constant inter-arrival time. We denote the period of stream S_i ($i = 1, 2, \ldots, n$) by P_i.
2. The *deadline* of a synchronous message is the end of the period in which it arrives. That is, if a message in stream S_i arrives at time t, then its deadline is at time $t + P_i$.[2]

[2] This assumption may be relaxed.

3. Messages from different streams are independent in that message arrivals do not depend on the initiation or the completion of transmission requests for other messages.
4. The *length* of each message in stream S_i is C_i which is the maximum amount of time needed to transmit this message.
5. Asynchronous messages are non-periodic and do not have explicit deadline requirements.

The *Utilization factor* of a synchronous message set, $U(M)$ is defined as the fraction of time spent by a ring in the transmission of the synchronous messages. That is,

$$U(M) = \sum_{i=1}^{n} \frac{C_i}{P_i} \tag{3}$$

where n is the number of synchronous message steams.

A subset of messages, denoted by M_C, are mission critical. That is,

$$M_C \subseteq M_A \cup M_B. \tag{4}$$

The objective of our study is to develop technology that guarantees the message deadlines of M_A and M_B under normal conditions, and guarantees the message deadlines of M_C when a network fault occurs. To facilitate this fault tolerant guarantee of mission critical messages, we assume that nodes which are required to transmit/receive a critical message are connected to both rings. In this way, once a fault occurs on one ring, another ring can be used to transmit/receive critical messages.

Without loss of generality we assume that there is one stream of synchronous messages on a node per ring (i.e., $m = n$). We can formally prove that an arbitrary token ring network where a node may have zero, one, or more streams of synchronous messages to transmit can be transformed into a logically equivalent network with one stream of synchronous message per node.

3 Synchronous Capacity Allocation

3.1 Timed Token MAC Protocol

Guaranteeing message deadlines requires the proper control of medium access. This is the function of a medium access control (MAC) protocol. FDDI uses the timed token MAC protocol in which messages are segregated into separate classes: the *synchronous* class and the *asynchronous* class [12]. Synchronous messages arrive at the system at regular intervals and may be associated with deadline constraints. The idea behind the timed token protocol is to control the token rotation time. During network initialization, a protocol parameter called the *Target Token Rotation Time* ($TTRT$) is determined which indicates the expected token rotation time. Each station is assigned a fraction of the $TTRT$, known as its *synchronous capacity*[3], which is the maximum time a station is permitted to transmit synchronous messages *every time* it receives the token. Thus, once a node receives the token, it transmits its synchronous messages, if any, for a time no more than its allocated synchronous capacity. It can then transmit its asynchronous messages only if the time elapsed since the previous token departure from the same node is less than the value of $TTRT$, i.e., only if the token arrives earlier than expected.

Guaranteeing a message deadline implies that the message will be transmitted before its deadline. With a token passing protocol, a node can transmit messages only when it captures the token. Hence, if a message deadline is to be guaranteed, the token should visit the node, where the message is waiting, before the expiration of the message deadline. That is, in order to guarantee message deadlines in a token ring network, it is *necessary* to bound the time between two consecutive visits of the token to a node (called the *token rotation time* or *access time*). The timed token protocol

[3] Some other synonymous terms that researchers use are: *Bandwidth allocation*, *Synchronous allocation*, *Synchronous bandwidth assignments*, and *High Priority token holding time*.

possesses this property. In [18, 39], Johnson and Sevcik formally proved that when the network operates normally (i.e, there is no failure), the token rotation time between two consecutive visits to a node is bounded by twice the expected token rotation time (i.e., $2 \cdot TTRT$).

Although the prerequisite of 'bounded token rotation time' is indispensable, it is however inadequate for guaranteeing message deadlines. A node with insufficient synchronous capacity may be unable to complete the transmission of a synchronous message before its deadline. On the other hand, allocating excess synchronous capacities to the nodes could increase the token rotation time, which may also cause message deadlines to be missed. Thus, guaranteeing message deadlines is also dependent upon the appropriate allocation of the synchronous capacities to the nodes.

3.2 Allocation Schemes

Definition and Examples

Denote H_i as the synchronous capacity of node i for a particular ring. The synchronous message parameters (given by the C_i's and P_i's) at the various stations, the value of $TTRT$, and the ring latency τ should be the dictating factors for the allocation of the H_i's. We define a synchronous capacity allocation scheme as an algorithm that, given as input the values of all C_i and P_i in the message set and the values of $TTRT$ and τ, will produce as output the values of the synchronous capacities H_i to be allocated to each station i in the network.

Let function f represent an allocation scheme. Then,

$$(H_1, H_2, \ldots, H_n) \;=\; f(C_1, C_2, \ldots C_n, P_1, P_2, \ldots P_n, TTRT, \tau). \tag{5}$$

Some of the allocation schemes which we consider are listed below:

- *Full length scheme.* In this scheme the synchronous capacity allocated to a node is equal to the total time required for transmitting its synchronous messages, i.e.,

  $$H_i = C_i. \tag{6}$$

  This scheme attempts to transmit a synchronous message arriving at a node in a single turn rather than splitting it into chunks and distributing its transmission evenly over its period P_i.

- *Proportional scheme.* In this scheme the synchronous capacity allocated to node i is proportional to the ratio of C_i and P_i at node i, i.e.,

  $$H_i = \frac{C_i}{P_i}(TTRT - \tau). \tag{7}$$

- *Equal partition scheme.* In this scheme the usable portion of $TTRT$ is divided equally among the n nodes in allocating their synchronous capacities, i.e.,

  $$H_i = \frac{TTRT - \tau}{n}, \tag{8}$$

  where n is the number of nodes in the system.

- *Normalized proportional scheme.* In this scheme the synchronous capacity is allocated according to the normalized load of the synchronous messages on a node, i.e.,

  $$H_i = \frac{C_i/P_i}{U}(TTRT - \tau), \tag{9}$$

  where $U = \sum_{i=1}^{n} C_i/P_i$.

- *Local scheme.* In this scheme, the message length is divided by the worst case number of token visits to a node during a single message period:

$$H_i = \frac{L_i}{\lfloor P_i/TTRT \rfloor - 1}. \tag{10}$$

Note that this scheme allocates the synchronous capacity without using information regarding messages on other nodes. This is advantageous for run-time network management. If the parameters of a message stream at a node change during run-time, a local allocation scheme need only adjust the synchronous capacity of the node involved. Other nodes are not disturbed. That is, the entire network can continue its normal operations while individual nodes change their synchronous capacities in response to changing message parameters.

Constraints

The synchronous capacities allocated to the nodes by any scheme must satisfy two constraints in order to ensure that real-time messages can be transmitted before their deadlines and that the timed token protocol requirements are satisfied.

- *Protocol Constraint*: The sum total of all the synchronous capacities allocated to all the nodes in the ring should not be greater than the target token rotation time minus the token walk time, i.e.,

$$\sum_{i=1}^{n} H_i \leq TTRT - \tau. \tag{11}$$

- *Deadline Constraint*: The allocation of the synchronous capacities to the nodes should be such that the synchronous messages are always guaranteed to be transmitted before their deadlines. In other words, if x_i is the minimum amount of time available for node i to transmit its synchronous messages in a time interval $(t, t + P_i)$, then

$$x_i \geq C_i. \tag{12}$$

Note that x_i will be a function of H_i and the number of token visits to node i in the time interval $(t, t + P_i)$.

Formally, we say that a set of synchronous messages is *guaranteed* by an allocation scheme if both the protocol and the deadline constraints are satisfied. Once a message set is guaranteed, messages will be transmitted before their deadlines as long as the network operates normally.

Performance Metric

Obviously, there are many ways to construct synchronous capacity allocation schemes. We would like to classify and evaluate allocation schemes so that proper recommendations can be made to network designers and managers on what allocation schemes to use. An appropriate metric must first be selected in order to evaluate and compare the effects of synchronous capacity allocation schemes on the performance of FDDI networks.

As mentioned earlier, we adopt the methodology developed in analyzing the rate monotonic scheduling algorithm. Following this methodology, we use the *Worst Case Achievable Utilization* as the metric to be used in evaluating and comparing the schemes.

We say U_x is an *achievable utilization* of scheme x if scheme x can guarantee all synchronous message sets whose utilization is less than or equal to U_x. The *Worst Case Achievable Utilization* U_x^* of a scheme x is the least upper bound of its achievable utilizations U_x. That is, as long as the utilization factor of a synchronous message set is not more than U_x^*, then the message set can

Name	*Formula of H_i*	*W.C.A.U.*[*]
Full length	$H_i = C_i$	0
Proportional	$H_i = \frac{C_i}{P_i} \cdot (TTRT - \tau)$	0
Equal partition	$H_i = \frac{TTRT-\tau}{n}$	$\frac{1-\alpha}{3n-(1-\alpha)}$
Normalized proportional	$H_i = \frac{C_i/P_i}{U} \cdot (TTRT - \tau)$	$\frac{1-\alpha}{3}$
Local	$H_i = \frac{C_i}{\lfloor P_i/TTRT \rfloor - 1}$	$\frac{1-\alpha}{3}$

[*] W.C.A.U. is the abbreviation of 'Worst Case Achievable Utilization'.

[*] $\alpha = \tau/TTRT$.

Table 1: Summary of the Synchronous Capacity Allocation Schemes.

be guaranteed by scheme x. We consider one scheme to be better than another if its Worst Case Achievable Utilization is higher.

The main advantages of using the Worst Case Achievable Utilization as the performance metric are as follows:

- This metric evaluates the predictability of a hard real-time communication system. As long as the utilization of a synchronous message set is within the bound specified by the metric, all synchronous messages in the set will meet their deadlines.
- This metric gives a measure of the stability of the system in the sense that the parameters of synchronous messages can be freely changed without affecting the deadline guarantees, provided that the total utilization of the message set is held within the limit.
- In practice, using this metric simplifies the network management considerably when configuring the system, as it eliminates the problem of being encumbered with individual values of synchronous and asynchronous message lengths, inter-arrival intervals, phase differences between message arrivals, relative positions of the nodes, token position at initialization, etc. As long as network managers can ensure that the total utilization of time-critical synchronous messages is no more than the Worst Case Achievable Utilization of the protocol, they can be assured that the message set will be transmitted with no deadlines being missed.

Evaluation Results

We analyzed five synchronous capacity allocation schemes based on their worst case achievable utilizations. The results are summarized in Table 1.

Our analysis reveals that an improper allocation of the synchronous capacities (such as by the full length scheme and the proportional scheme) could lead to a Worst Case Achievable Utilization of 0%. That is, the deadline of some message could be missed even when the synchronous traffic is extremely low. Both the normalized proportional allocation scheme and the local allocation scheme, on the other hand, have a worst case achievable utilization of 0.33. If the utilization of a set of synchronous message streams is less than 0.33 of the usable network capacity, then the synchronous messages will be guaranteed by these allocation schemes. The remaining 0.67 of the usable network capacity can be used for the transmission of asynchronous messages.

4 Dealing with Link Faults

The results presented in the last section are based on the assumption that there is no network failure. To provide deadline guarantees in the presence of a network fault, we also have to exploit the dual ring architecture and the connection management mechanism proposed in the FDDI standard.

4.1 Dual Ring Architecture and Link Faults

The basic configuration of an FDDI network is a dual counter-rotating ring as shown in Figure 1. The dual rings provide fault tolerant properties to FDDI networks, since the existence of a link fault can be signalled on the opposing link. A link fault is defined as a fault that occurs on the links between nodes resulting in a lack of communication across a single fiber. Examples include a single broken fiber, a faulty optical receiver, and a faulty optical transmitter. Other faults (e.g., loss of power to a node) may be treated similarly to a link fault. See [29] for a survey of FDDI fault classification and management.

In the Station Management (SMT) of FDDI, there are built-in mechanisms to detect and to recover from a link fault. According to the FDDI standard, once a link fault is detected, a sequence of ring recovery processes (i.e., the token reclaim process, beacon process, etc.) will be initiated. If the fault is transient and hence recoverable, the ring may be functioning again after these processes. If the fau' is permanent, two additional approaches are specified by the FDDI standard to recover the network:

- *Wrap-up:* The fault domain is traced and the stations around the broken link perform a wrap-up operation, i.e., two rings are effectively connected to each other at the stations immediately adjacent to the fault. This re-establishes a single ring between all the nodes (see Figure 2).

- *Global Hold:* Another strategy is to prevent the wrap-up of the rings and hold the operational ring, as it is, for continuing communication service. The messages from the faulty ring can be transferred to the operational ring (see Figure 3).

4.2 Approaches

Although the connection management of FDDI guarantees network service *before* or *after* a single link fault occurs, it does not support transmission of messages on the faulty ring *during* fault detection and recovery. For hard real-time communication, this is inadequate because the fault detection and recovery processes take several seconds or more to complete. Message deadlines in many hard real-time applications are usually of a much smaller order of magnitude.

Furthermore, once a fault occurs on one ring, messages can only be transmitted on another ring. If both rings are fully utilized before the fault occurs, it is impossible to transmit over a single ring the messages that were previously on the two rings. Some of the messages will have to be dropped. We assume that when a link fault occurs, the network changes into a *link fault mode*. In this mode, not all the messages are to be transmitted. Only a subset of messages that are critical to the mission will be transmitted and their deadlines have to be guaranteed at any time, including during the period of fault detection and recovery. We assume that the capacity of one ring is sufficient to transmit these mission critical messages. The objective is to develop network run-time control schemes that will be able to guarantee the deadlines of critical messages through the entire mission even in the presence of a link fault.

The following approaches have been proposed to deal with this problem:

- *Full Duplication Method.* Duplicate the transmission of critical messages on both rings so that when one ring is unavailable, the deadlines of messages are still guaranteed because they are also transmitted on the other ring. This solution is the simplest but it suffers by wasting bandwidth during times when there is no fault.

- *Dynamic Reallocation Method.* With this approach, once a fault is detected, critical messages from the faulty ring are reallocated to another ring that is still operating. Although the

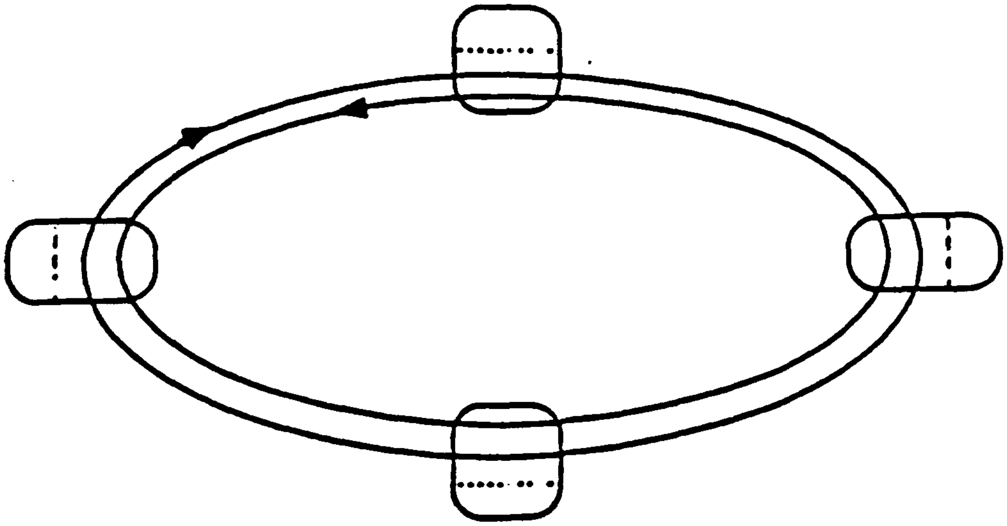


Fig 1: Dual Ring Architecture of FDDI

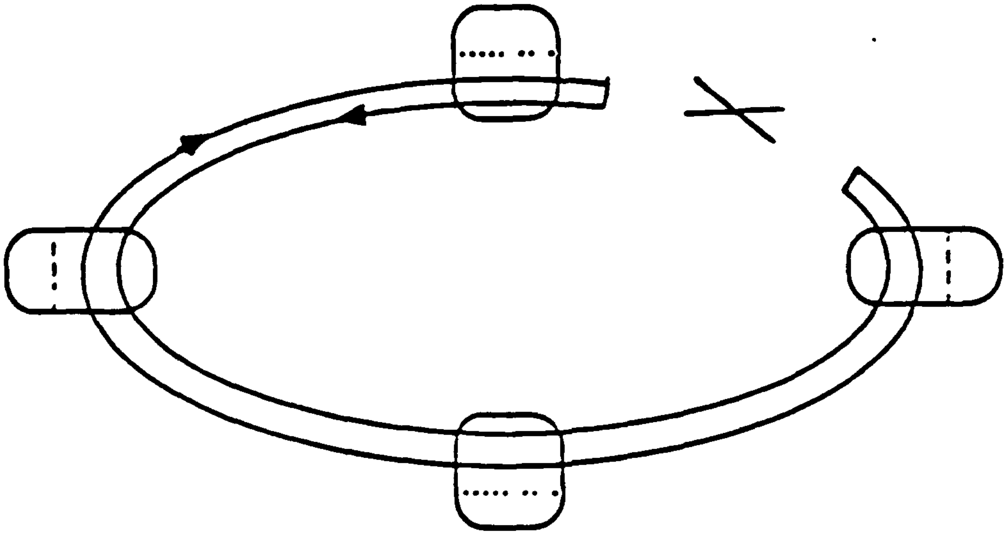


Fig 2: Fault Recovery after Wrapped Up

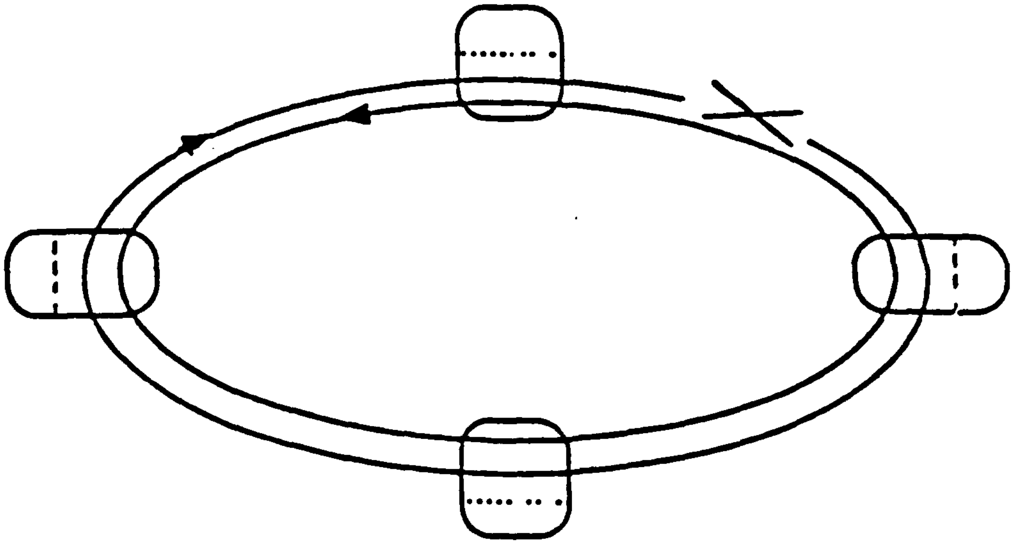


Fig 3: Fault Recovery with the Hold Policy

bandwidth is fully utilized when there is no fault in the network, the implementation of this approach requires a detailed analysis of timing factors in fault detection and mode change. A time efficient message reallocation scheme and mode change protocol are also needed. Furthermore, this solution cannot be applied to those applications where the deadlines of critical messages would be too small to tolerate the overhead of fault detection and dynamic reallocation.

- *Integrated Method.* An alternative is to integrate the full duplication method and the dynamic reallocation method. The transmission of critical messages with very small deadlines is duplicated on both rings. A dynamic reallocation will be performed for other critical messages when a link fault occurs on one ring. This method should utilize the network better than the full duplication method while overcoming the shortcomings of dynamic reallocation.

We are currently developing techniques for the implementation of the above three approaches, and to evaluate and compare the performance of these approaches. The performance metrics we are interested in include the effectiveness of network utilization in both normal and faulty situations, the run-time overheads, and the domain of applicable applications.

5 Summary

We address issues pertaining to deadline guarantees in a degraded FDDI network. We aimed at providing deadline guarantees to a set of mission critical messages throughout the entire mission, even in the presence of a fault. This is particularly important in practice because some critical applications do need non-interrupted real-time service.

Our approach is (upward) compatible with the proposed standard. Hence, the results obtained from our work will be immediately applicable to the design and analysis of distributed hard real-time systems where an FDDI network is used.

We analyze the system by deriving its worst case utilization bound. This metric is particularly important because it indicates the safety margin of the system and provides a measure of system stability. All previous work regarding this measure is related to the rate monotonic scheduling algorithm. Our work is the very first which derives the worst case utilization bound for a scheduling environment where global priority arbitration is not supported and hence the rate monotonic algorithm cannot be used.

References

[1] ANSI/IEEE Standard 802.4 - 1990 *Token passing bus access method and physical layer specifications*, The Institute of Electrical and Electronic Engineers, Inc., New York, 1990.

[2] ANSI Standard X3.139-1987, *FDDI Token ring media access control*, Feb. 28. 1986.

[3] ANSI Standard X3T9/90. *FDDI Token ring station management*, May 1990.

[4] B. Chen, A. Agrawal, and W .Zhao, "Optimal Synchronous Capacity Allocation for Hard Real-Time Communications with the Timed Token Protocol", *Proc. IEEE Real-Time Systems Symposium*, Dec. 1992.

[5] G. Agrawal, B. Chen, and W. Zhao, "Local synchronous capacity allocation schemes for guaranteeing message deadlines with timed token medium access control protocol", *Proc. IEEE Conf. Computer Communications, INFOCOM '93*, March 1993.

[6] G. Agrawal, B. Chen, W. Zhao, and S. Davari, "Guaranteeing synchronous message deadlines in high speed token ring networks with timed token protocol", *Proc. of IEEE International Conference on Distributed Computing Systems*, June 1992.

[7] L. Bergman, "Optical protocols for advanced spacecraft networks," *Proceedings of the space station evolution symposium*, Vol. 1, part 2, League city, Texas, Aug. 6-8, 1991.

[8] E. Chevers, "Advanced DMS architectures," *Proceedings of Space Station Evolution symposium*, Vol. 1, part 2, League city, Texas, Aug. 6-8, 1991.

[9] M. D. Cohn, "A network architecture for advanced aircraft," *Proc. IEEE Conf. on Local Computer Networks*, pp. 358-364, Minneapolis MN, Oct. 10-12, 1989.

[10] S. Davari and W. Zhao, "RMS aids real-time scheduling", *RJCIS Review*, Vol. 3, No. 1, 1991.

[11] D. T. Green and D. T. Marlow, "SAFENET - A LAN for navy mission critical systems," *Proc. 14th Conf. on Local Computer Networks*, Minneapolis, MN, pp. 340-346, Oct. 1989.

[12] R. M. Grow, "A timed token protocol for local area networks", *Proc. Electro/82, Token Access Protocols*, May 1982.

[13] R. M. Grow, "FDDI follow-on status," *Proc. IEEE Conf. on Local Computer Networks*, pp. 45-48, Minneapolis MN, Sept. 30 - Oct. 3, 1990.

[14] V. Iyer *and* S. P. Joshi, "New standards for local networks push upper limits for lightwave data," *Data communications.*, pp. 127-138, July 1984.

[15] V. Iyer and S. P. Joshi, "FDDI's 100 M-bps protocol improves on 802.5 Spec's 4-M-bps limit," *Electrical Design News . pp. 151-160*, May 2, 1985.

[16] R. Jain, " Error characteristics of fiber distributed data interface (FDDI)", *IEEE Trans. on Commun.*, vol. 38, No. 8, Aug. 1990.

[17] R. Jain, "Performance analysis of FDDI token ring networks: effect of parameters and guidelines for setting TTRT," *IEEE LTS*, pp. 16-22, May 1991.

[18] M. J. Johnson, "Proof that timing requirements of the FDDI token ring protocols are satisfied", *IEEE Trans. Commun.* vol. COM-35.no. 6, pp. 620-625, June 1987.

[19] L. Kleinrock. Distributed systems. *Communications on the ACM*, 28(11), November 1985.

[20] R. J. Kochanski and J. L. Paige, "SAFENET - The standard and its application," *IEEE LCS*, Vol. 2, No. 1., pp. 46-51, Feb. 1991

[21] J. F. Kurose, M. Schwartz, and T. Yemini, "Controlling window protocols for time-constrained communication in a multiple access environment," *Proc. IEEE Int. Data Communication Symp.* 1983.

[22] C. C. Lim, L. Yao, and W. Zhao, "A comparative study of three token ring protocols for real-time communications", *IEEE Conf. on Distributed Computing Systems*, pp. 308-317, Arlington, Texas, May 1991.

[23] C. L. Liu and J. W. Layland, "Scheduling algorithms for multiprogramming in a hard real time environment," *J. ACM*, Vol. 20, no. 1, 1973, pp. 46-61.

[24] N. Malcolm and W. Zhao, "Version selection schemes for hard real-time communications.", *Proc. IEEE Real-time Systems Symposium*, San Antonio, TX, Dec., 1991.

[25] J. Mccool, " FDDI - getting to the inside of the ring", Data Commun., pp. 185-192, March 1988.

[26] MIL-HDBK-818-1, "Survivable adaptable fiber optic embedded network", Oct. 1992.

[27] J. Ng and J. Liu, "Performance of local area network protocols for hard real-time applications", *IEEE Conf. on Distributed Computing Systems*, pp. 318-326, Arlington, Texas, May 1991.

[28] K. B. Ocheltree and R. M. Montalvo, "FDDI ring management," *Proc. IEEE Conf. on Local Computer Networks*, pp. 18-23, Minneapolis MN, Oct. 10-12, 1989.

[29] K. B. Ocheltree, "Using redundancy in FDDI networks," *Proc. IEEE Conf. on Local Computer Networks*, pp. 261-267, Minneapolis MN, Sept. 30 - Oct. 3, 1990.

[30] Lt. J. L. Paige, "SAFENET - A navy approach to computer networking," *Proc. IEEE Conf. on Local Computer Networks*, pp. 268-273, Minneapolis MN, Sept. 30 - Oct. 3, 1990.

[31] J. Pang and F. A. Tobagi, "Throughput analysis of a timer controlled token passing protocol under heavy load", *IEEE Trans. on Communications*, Vol. 37, No. 7, pp. 694-702, July 1989.

[32] K. Ramamritham, J. Stankovic, and W. Zhao. Meta-level control in distributed real-time systems. In *Proceedings of IEEE Seventh International Conference on Distributed Computing Systems*, pages 10 - 17, September 1987.

[33] K. Ramamritham, J. Stankovic, and W. Zhao. Distributed scheduling of tasks with deadlines and resource requirements. *IEEE Transactions on Computers*, 38(8), August 1989.

[34] F. E. Ross, "FDDI—A tutorial," *IEEE Commun. Mag.*, vol. 24, no. 5, pp. 10-17, 1986.

[35] F. E. Ross, " Rings are round for good", *IEEE Network Mag.*, J›n. 1987.

[36] F. E. Ross, "An overview of FDDI: The fiber distributed data interface," *IEEE Journal on Sel. Areas in Comm.*, pp.1043-1051, Vol. 7, Sept. 1989.

[37] SAE, Aerospace Systems Division, Committee AS-2, "Linear token-passing multiple data bus," AS4074.1, Version 4.0, Jan. 25, 1988.

[38] SAE, Aerospace Systems Division, Committee AS-2, "High speed ring bus (HSRB)," AS4074.2, Jan. 27, 1988.

[39] K. C. Sevcik and M. J. Johnson, "Cycle time properties of the FDDI token ring protocol," *IEEE trans. Software Eng.*, Vol. SE-13, No. 3, pp. 376-385, 1987

[40] L. Sha and J. B. Goodenough, "Real-time scheduling theory and Ada*," *IEEE Computer*, April 1990., pp. 53-62.

[41] K. G. Shin and C. Hou, "Analytic evaluation of contention protocols used for real-time systems," *Proc. IEEE Real-Time Systems Symp.*, Dec 1990.

[42] K. G. Shin and Q. Zheng. Mixed time-constrained and non-time-constrained communications in local area networks. To appear in IEEE Transactions on Communications.

[43] R. Southard, "Fiber optics: A winning technology for LANs," *Electronics*, pp. 111-114, Feb. 1988.

[44] J. Stankovic. A perspective on distributed computer systems. *IEEE Transactions on Computers*, C-33(12), December 1984.

[45] J. K. Strosnider, J. Lehoczky and L. Sha. "Advanced real-time scheduling using the IEEE 802.5 token ring," Proc. IEEE Real-Time Systems Symp., pp. 42-52, Dec 1988.

[46] R. W. Uhlhorn, "The fiber-optic high-speed data bus for a new generation of military aircraft," *IEEE LCS*, Vol. 2, No. 1, pp. 36-45, Feb. 1991.

[47] J. N. Ulm, "A timed token ring local area network and it's performance characteristics," *Proc. Conf. Local Computer Networks,* Feb. 1982, pp. 50-56.

[48] A. M. van Tilborg and G. M. Koob. *Foundations of Real-Time Computing: Formal Specifications and Methods.* Kluwer Adademic Publishers, 1991.

[49] A. M. van Tilborg and G. M. Koob. *Foundations of Real-Time Computing: Scheduling and Resource Management.* Kluwer Academic Publishers, 1991.

[50] A. C. Weaver and R. Simoncic, "Communications for the NASA space station," *Proc. IEEE Conf. on Local Computer Networks*, pp. 333-346, Minneapolis MN, Oct. 10-12, 1989.

[51] L. Yao and W. Zhao, "Performance of an extended IEEE 802.5 protocol in hard real-time systems.", *Proc. IEEE Conf. Computer Communications, INFOCOM '91*, April 1991.

[52] S. Yau. Special Issue on Distributed Computer Systems. In *IEEE Transactions on Computers*, Vol. 38, 1989.

[53] W. Zhao, editor. *Special Issue on Real-Time Operating Systems, ACM Operating System Review*, volume 23. ACM Press, 1989.

[54] W. Zhao and K. Ramamritham, "Virtual time CSMA protocols for hard real-time communications", *IEEE Transactions on Software Engineering*, SE-13(8):938-952, Aug 1987.

[55] W. Zhao, K. Ramamritham, and J. Stankovic. Scheduling tasks with resource requirements in hard real-time systems. *IEEE Transactions on Software Engineering*, SE-13(5):564–577, May 1987.

[56] W. Zhao, K. Ramamritham, and J. A. Stankovic. Preemptive scheduling under time and resource constraints. *IEEE Transactions on Computers*, C-36(8):949 – 960, August 1987.

[57] W. Zhao, J. Stankovic, and K. Ramamritham. A window protocol for transmission of time constrained messages. *IEEE Transactions on Computers*, C-39(9):1186 – 1203, September 1990.

List of Transparency Sets from Workshop

(Not all transparency sets were made available by workshop participants)

Transparencies from Workshop Presentations

(Not all transparencies were made available by workshop participants)

I. Overview of Maruti

II. The Maruti-Mach Initiative

Maruti-2 on Mach 3.0

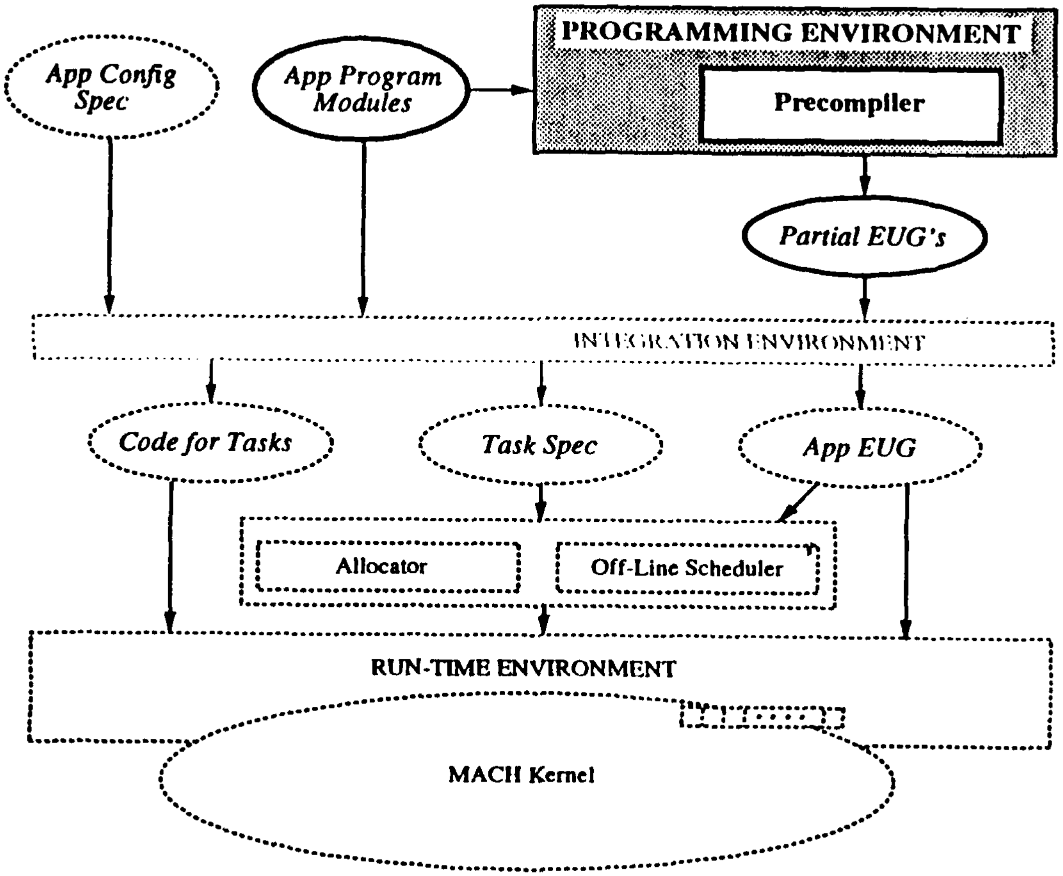
Why Mach?

- Separates policies from resources
- Versatile thread control
- Fast message-passing
- Uniform treatment of synchronization & communication (messages)
- Scalable to distributed environment
- Platform for Unix development

Related Projects

- Real-Time Mach

Programming Environment



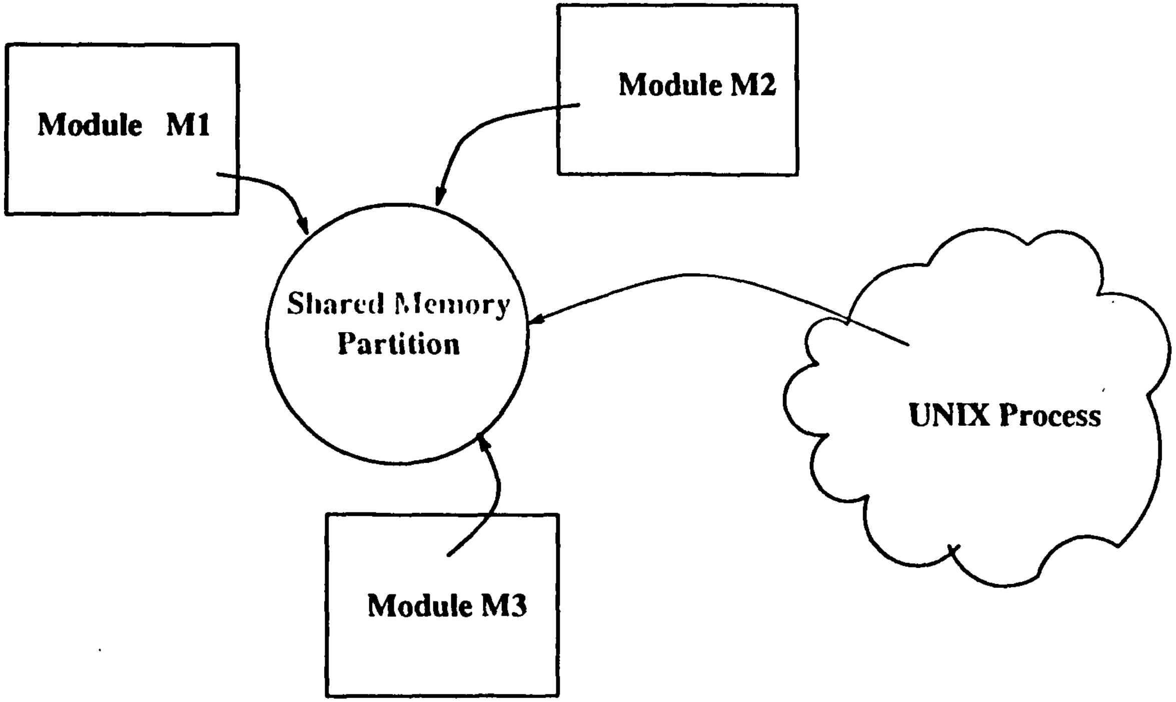
Maruti Programming Language

- Supports **modular** and **distributed programming**.
- Supports **static scheduling**.
- Supports **time** and **resource requirement estimation**.

Modular Programming

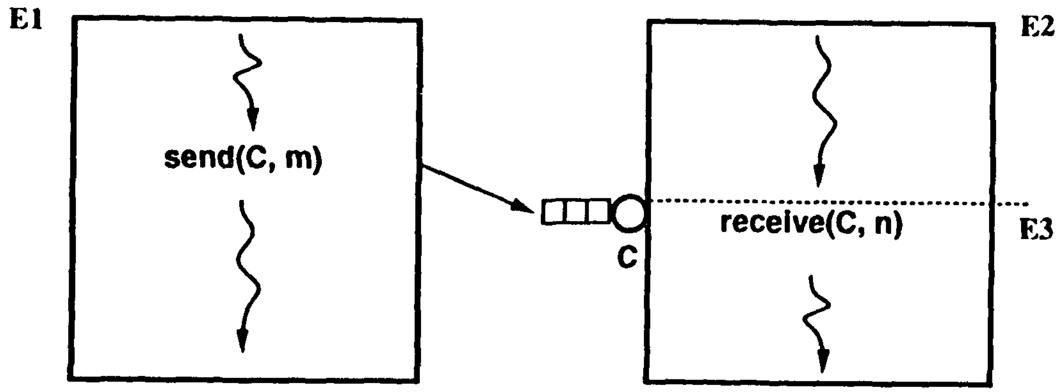
- A **module** consists of
  - several source files,
  - an interface.
- An interface provides declaration of public names.
- A module becomes a Mach **task**.
- Communication means between tasks:
  - shared memory partitions,
  - message passing.

Inter-Task Sharing



Message Passing,

- Successful message transfers are guaranteed by
  - **prealloction** of message buffer space,
  - **prescheduling** of senders and receivers.



$E1 \prec E3$

Message Passing(Cont'd)

- Simple Primitives

(1) **`send(C, m)`**

`send()` is a simple *message ccpy-out* operation.

(2) **`receive(C, m)`**

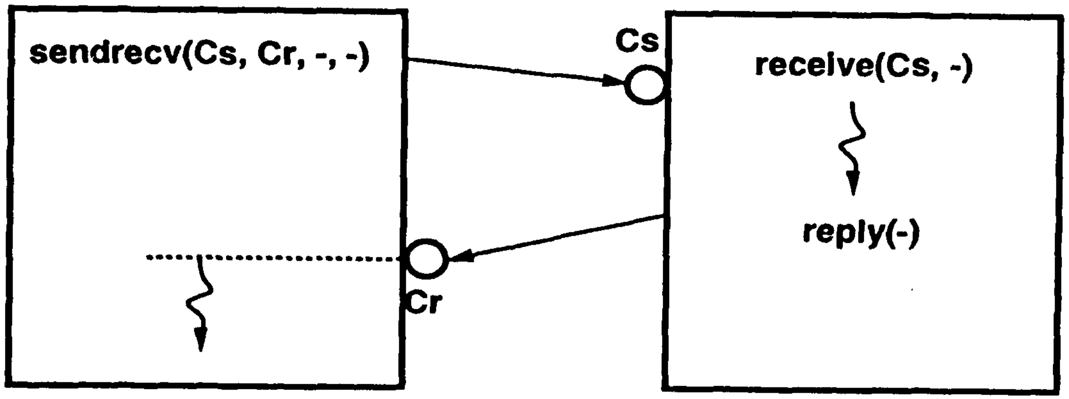
`receive()` is a simple *message copy-in* operation.

Message Passing(Cont'd)

- RPC Primitives

  (1) `sendrecv(Cs, Cr, ms, mr)`

  (2) `reply(m)`



Message Passing(Cont'd)

- Multiple receives as a **joiner**

```
join
{
   on receipt(C1, m1): statement1;
   on receipt(C2, m2): statement1;
                 . . . .
   on receipt(Ck, mk): statement1;
   exception:          ex_handler;
}
```

Program Decomposition

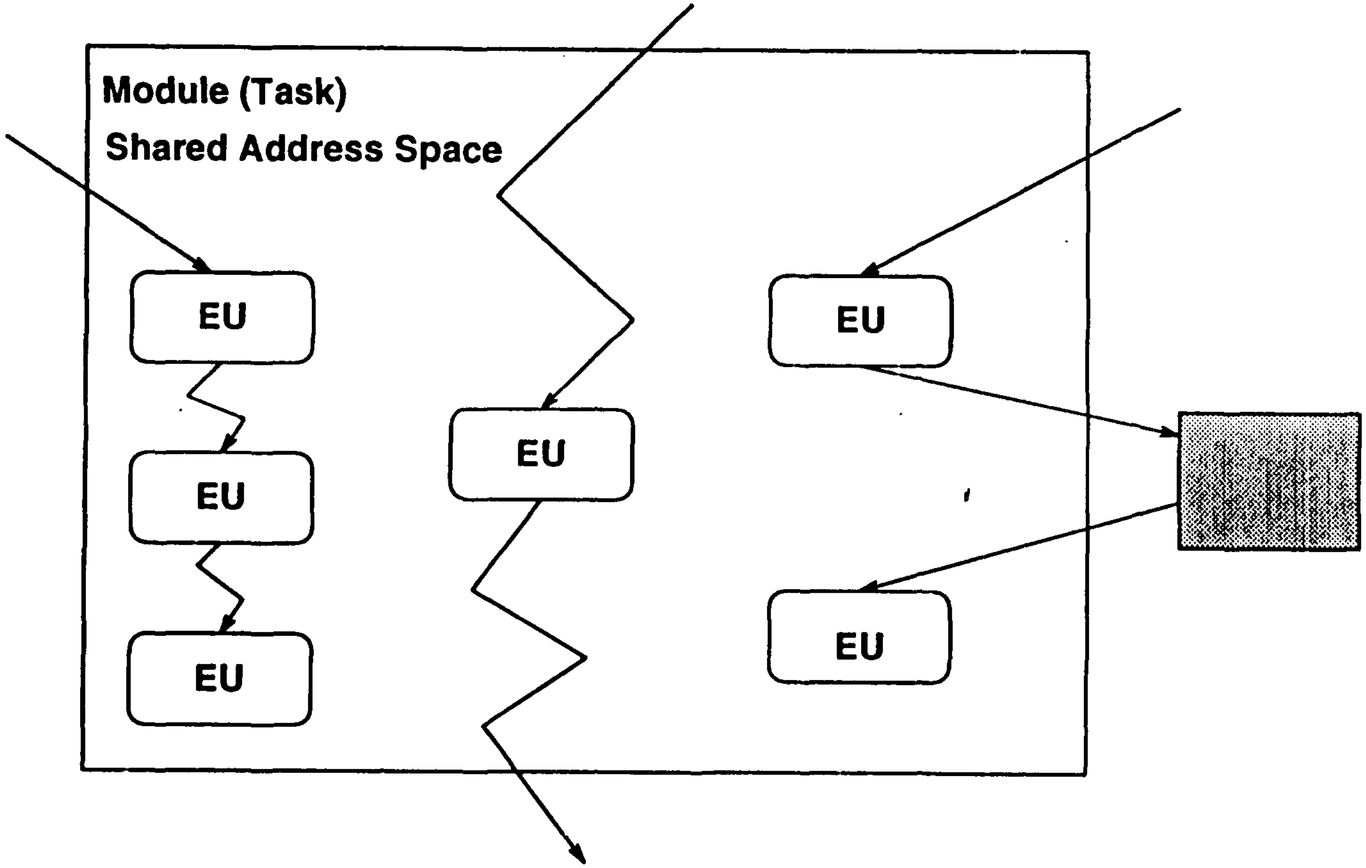
- An **elementary unit (EU)** is
  - a schedulable unit expected by Maruti scheduler,
  - a non-preemptable section of code.
- EU's have three types of constraints.

(1) **Precedence** imposed by control flow and message transfer.

(2) **Timing constraint** imposed by timing specification.

(3) **Mutual Exclusion** imposed by critical sections.

Basic Application Structures



Program Decomposition(Cont'd)

- **EU Graph** EUG = (N, E)

  (1) N =a set of EU's in a program.

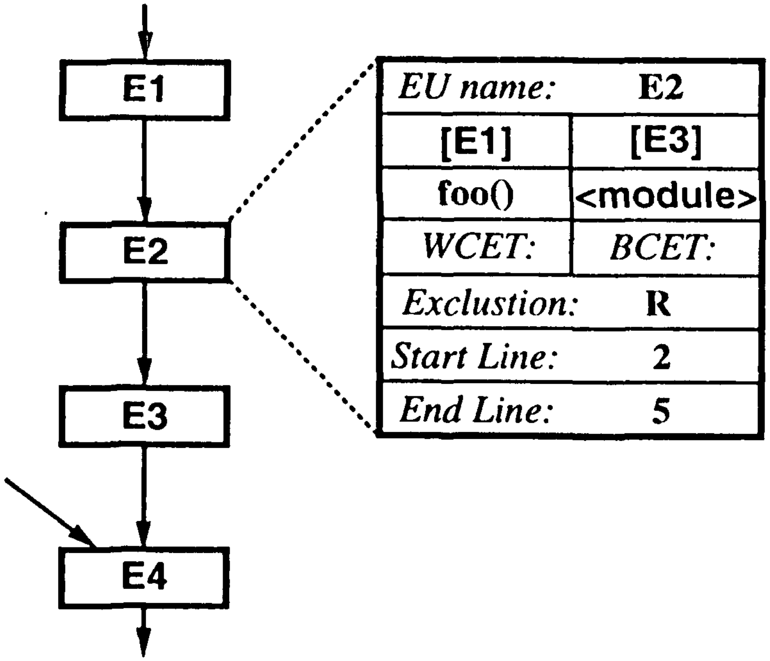
  (2) E =a set of precedence constraints between EU's.

- The EUG is the intermediate representation of an application in Maruti.

Program Decomposition(Cont'd)

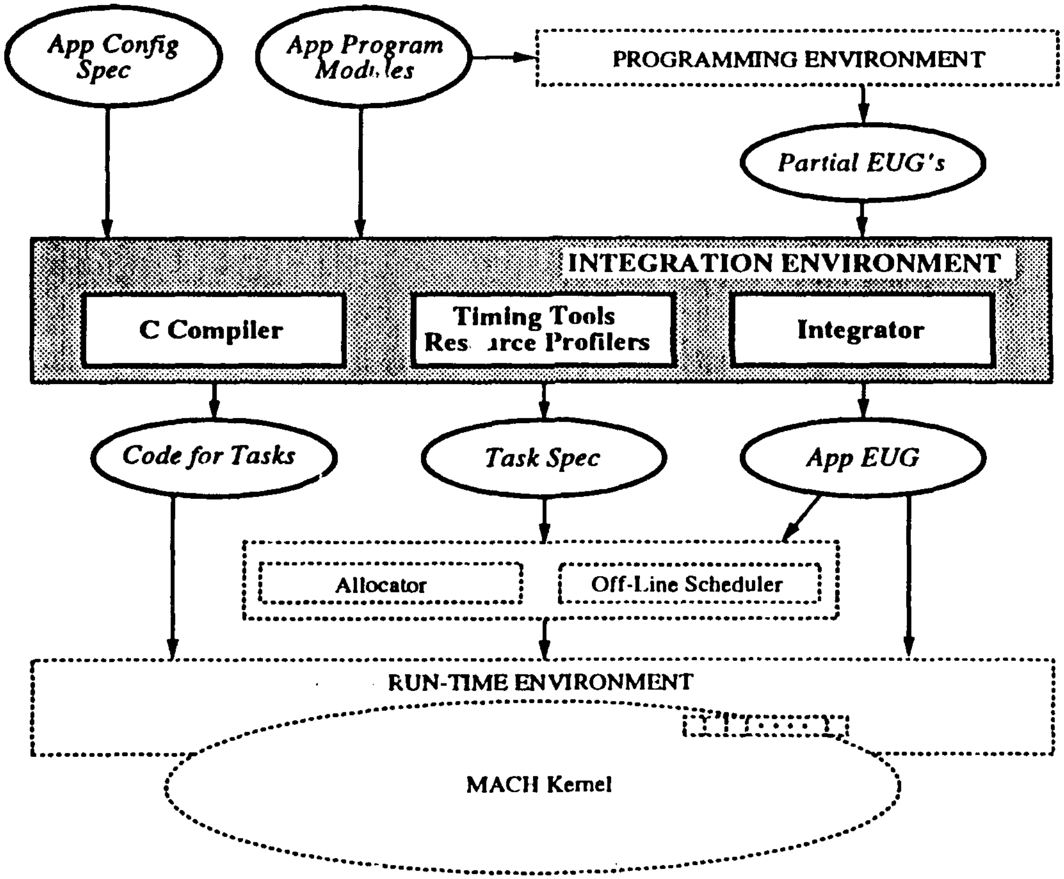
Partial EUG Example

```
E1   1:  S1;
     ..................................
E2   2:  region( R )
     3:  {
     4:      foo();
     5:  }
     ..................................
E3   6:  S3;
     ..................................
E4   7:  receive(C1, m);
     8:  S4;
     9:  send(C2, n);
```

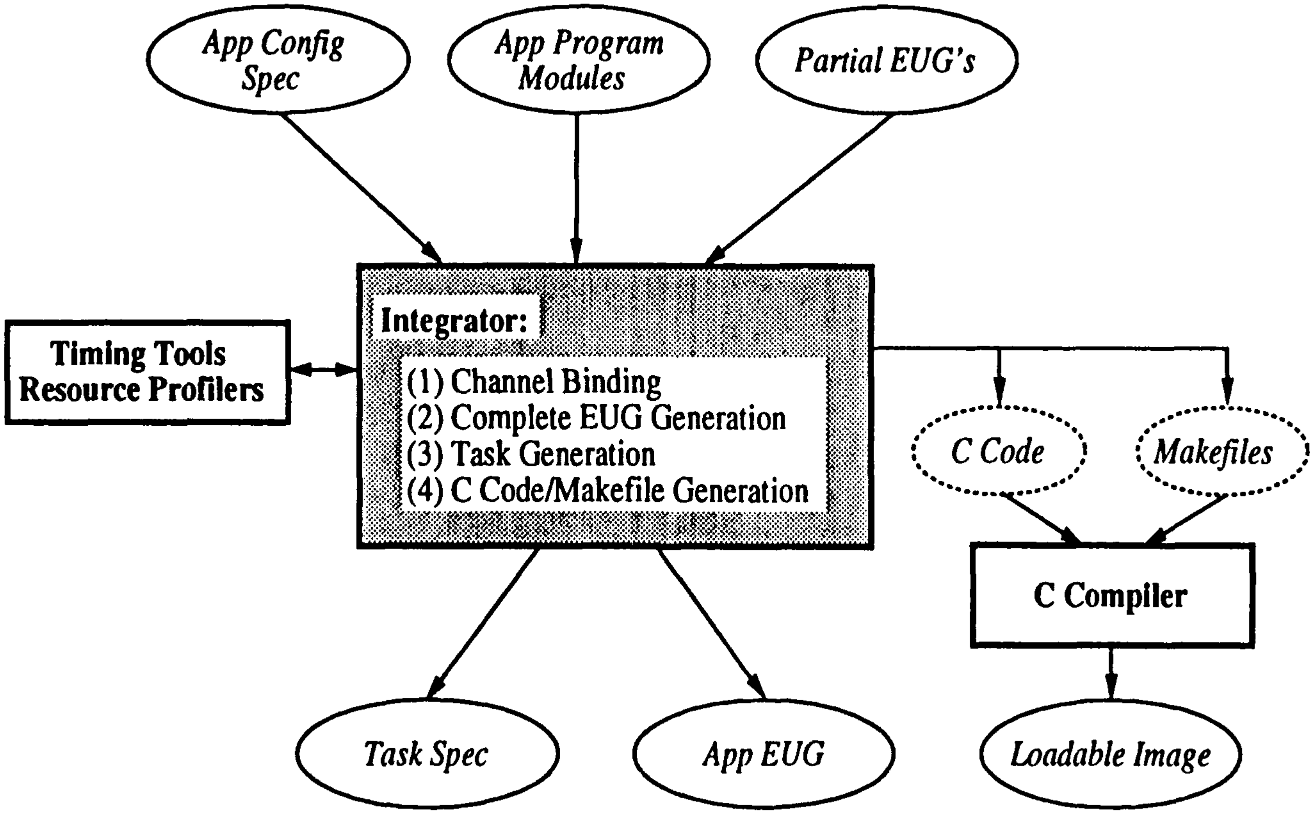


EU name:	E2
[E1]	[E3]
foo()	<module>
WCET:	*BCET:*
Exclustion:	R
Start Line:	2
End Line:	5

Integration Environment



Integration Process



App Config Spec

App Program Modules

Partial EUG's

Timing Tools
Resource Profilers

Integrator:

(1) Channel Binding
(2) Complete EUG Generation
(3) Task Generation
(4) C Code/Makefile Generation

C Code

Makefiles

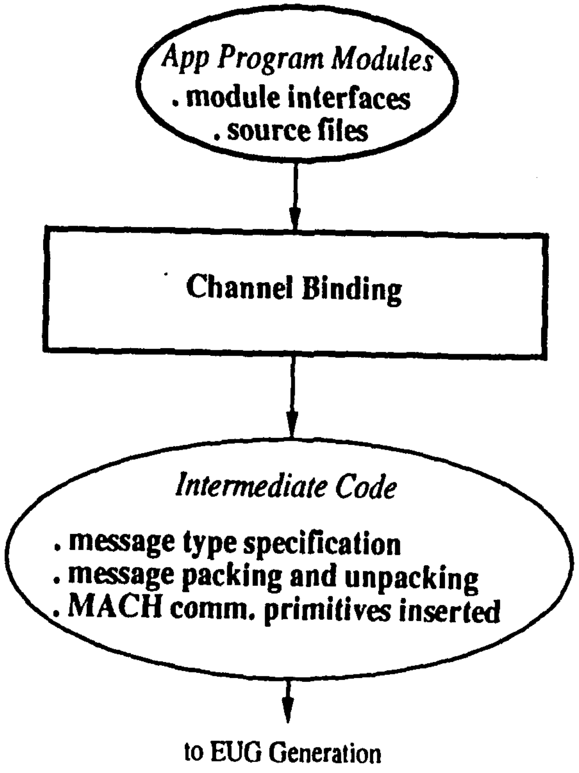
C Compiler

Task Spec

App EUG

Loadable Image

Channel Binding



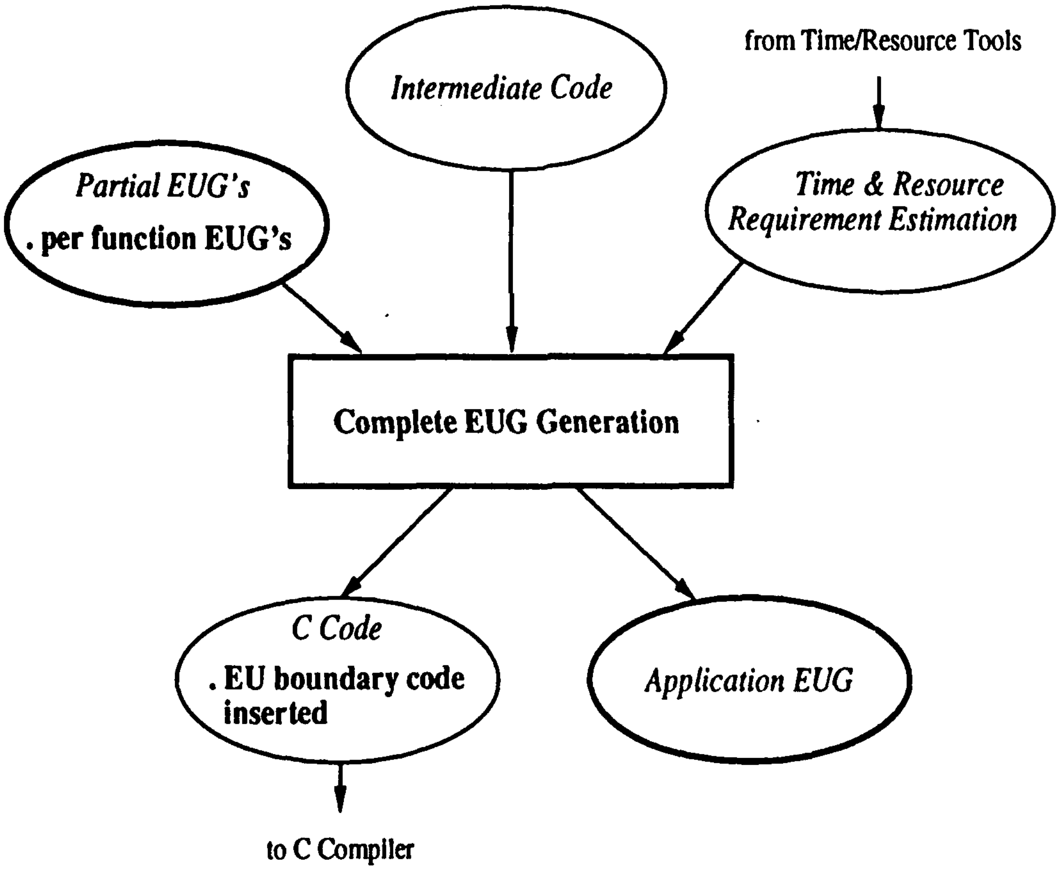
App Program Modules
- module interfaces
- source files

Channel Binding

Intermediate Code
- message type specification
- message packing and unpacking
- MACH comm. primitives inserted

to EUG Generation

Application EUG Generation



Partial EUG's
. per function EUG's

Intermediate Code

from Time/Resource Tools

Time & Resource Requirement Estimation

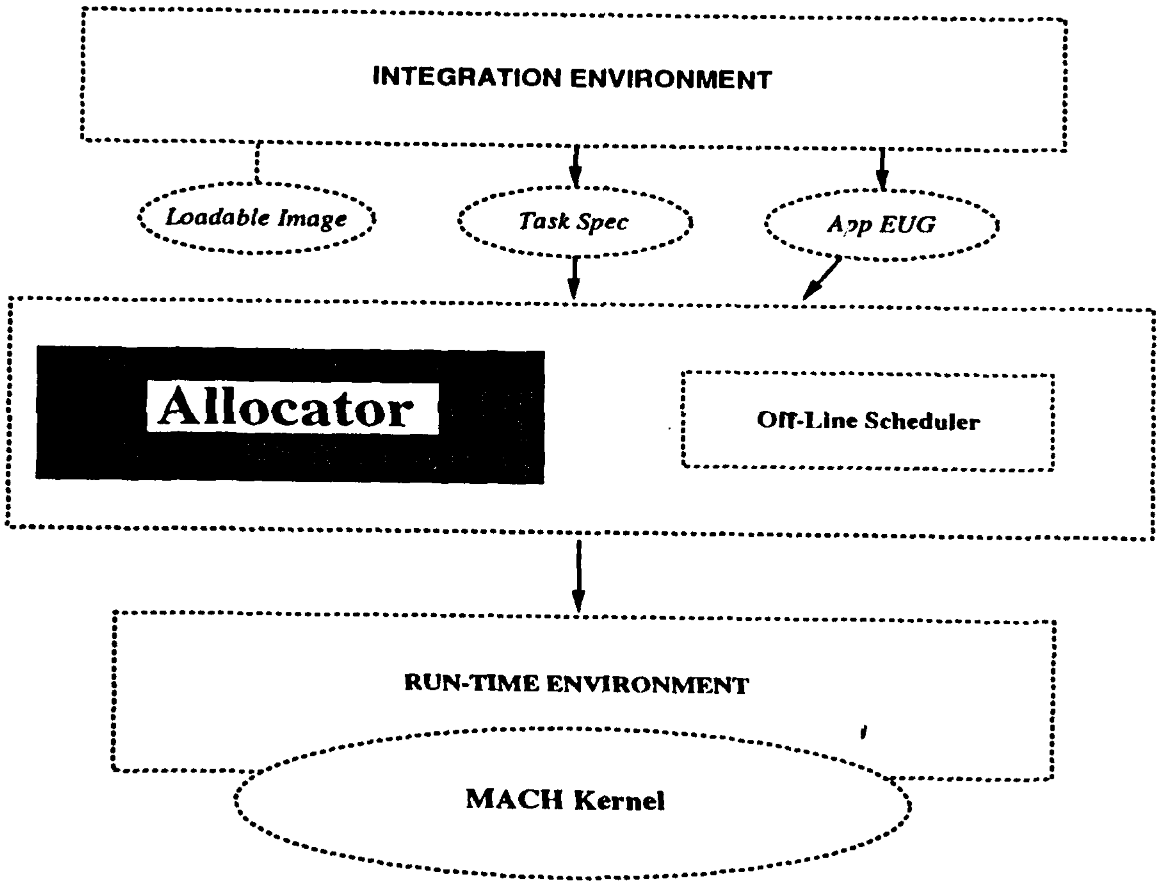
Complete EUG Generation

C Code
. EU boundary code inserted

to C Compiler

Application EUG

Allocator



INTEGRATION ENVIRONMENT

Loadable Image

Task Spec

App EUG

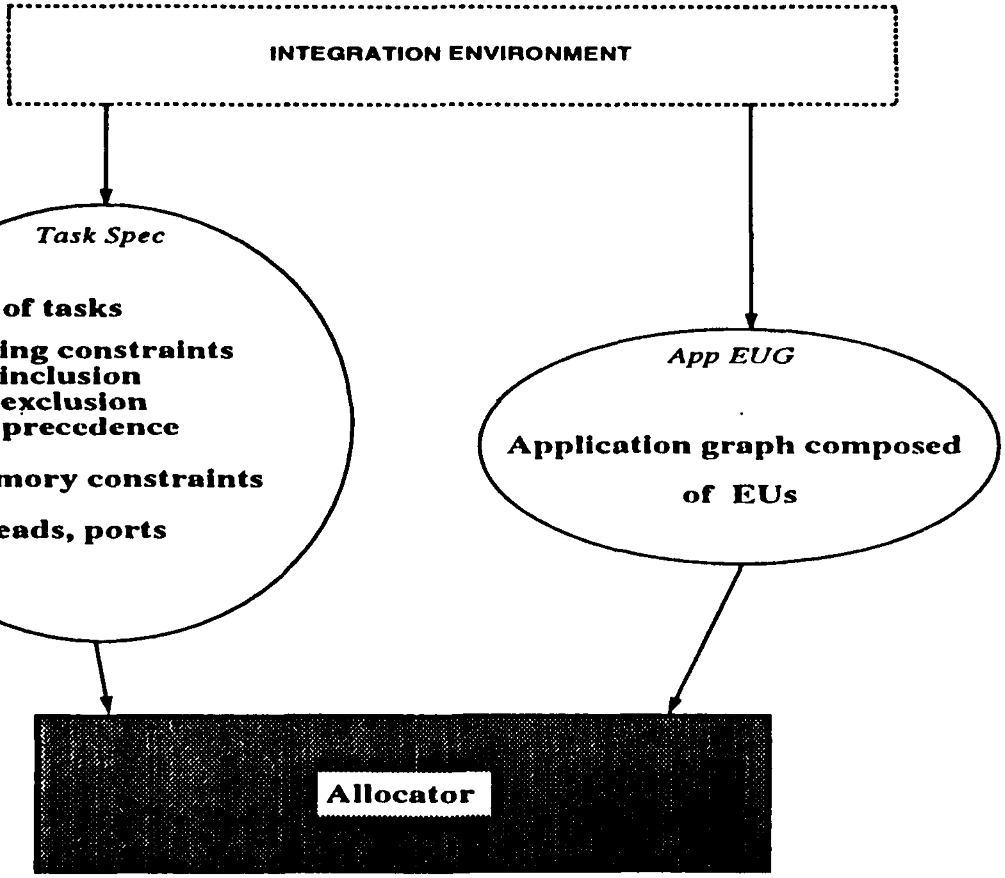
Allocator

Off-Line Scheduler

RUN-TIME ENVIRONMENT

MACH Kernel

Application model



INTEGRATION ENVIRONMENT

Task Spec

- list of tasks
- timing constraints
  e.g inclusion
  exclusion
  precedence
- memory constraints
- threads, ports

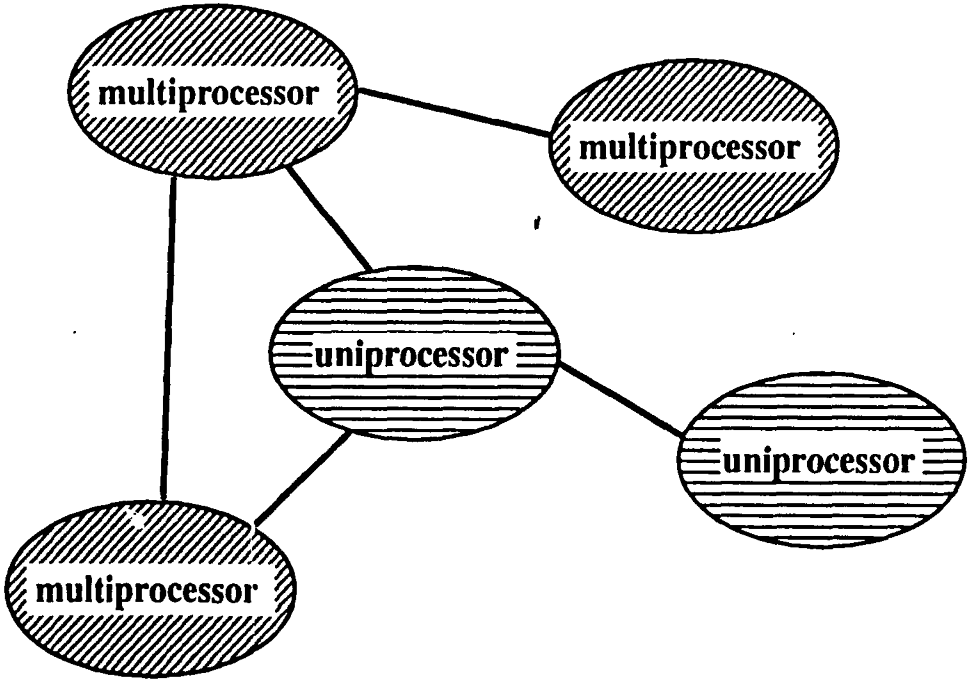
App EUG

Application graph composed of EUs

Allocator

Resources

- processors
- ports
- memory



Allocation Scheme

- Global Allocation
  - select “best” node on which to schedule the task
  - maps tasks to that node
- Local Allocation
  - allocation within a node
  - interaction with the scheduler

Scheduling

- Time-Driven
- Non-Preemptive
- Cyclic (LCM of periodic tasks)
- Offline Scheduler
  - Analyzes task set to reduce search space
  - Uses Heuristics to create a schedule
- Online Scheduler
  - Dispatches tasks from the schedule
  - Accepts/Rejects new tasks

Scheduling: Model

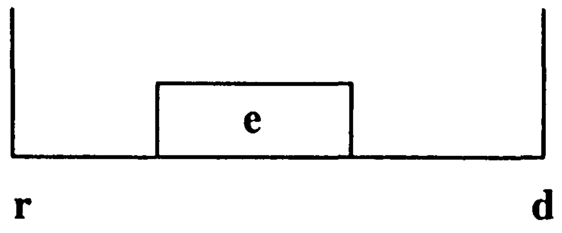
A set of jobs $\Gamma = \{\tau_i : 1 \leq i \leq n\}$, where $\tau_i = [r_i, e_i, d_i]$

Notation:

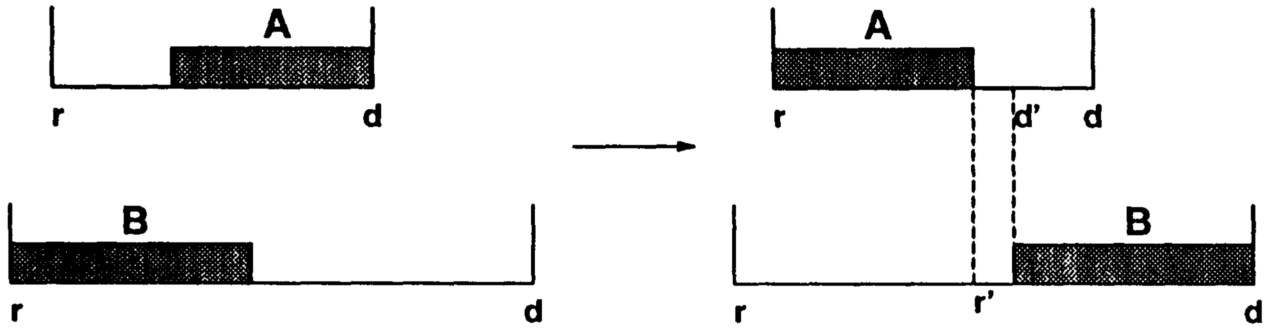
- r_i release time
- e_i execution time
- d_i deadline
- s_i start time
- f_i finish time

Constraints:

- $s_i \geq r_i$
- $f_i \leq d_i$



Scheduling: Approach



B cannot execute before A
(Precedence (A -> B))

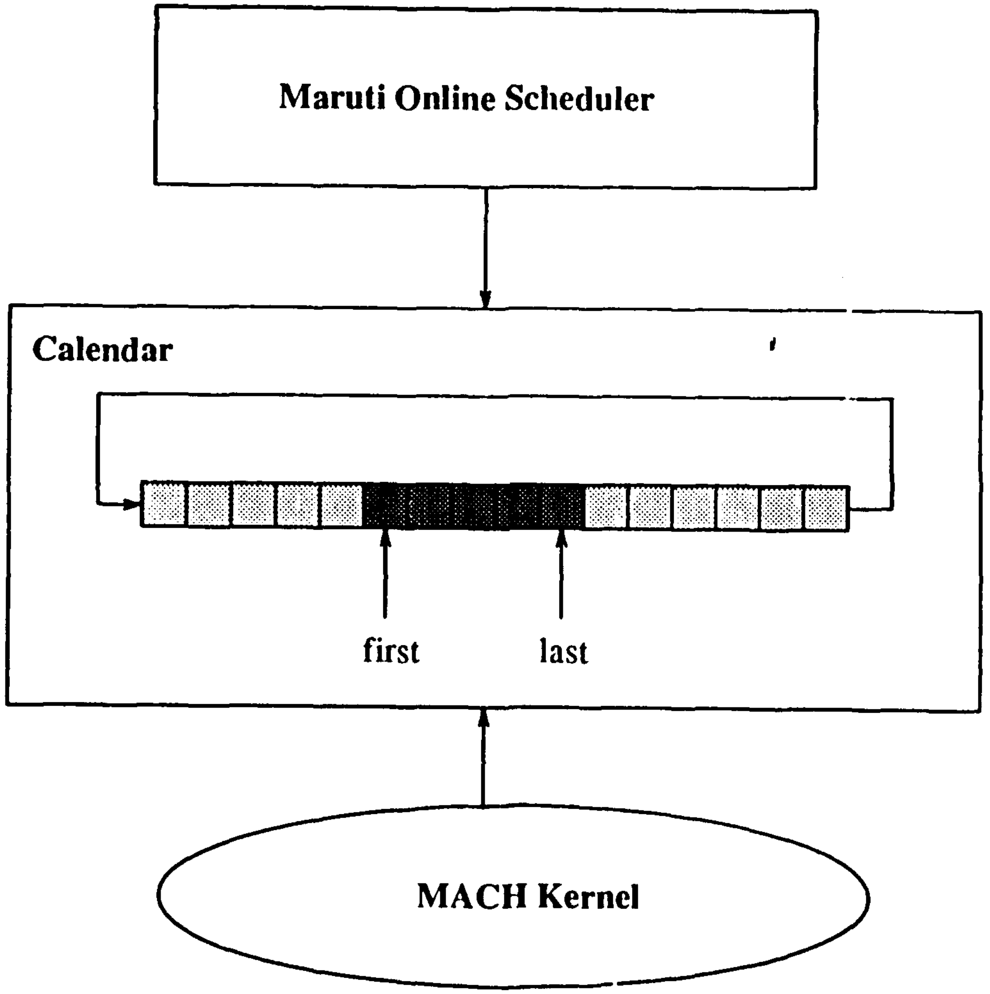
Modified Ready Time and Deadline
(Window Modification)

- Pairwise Schedulability Analysis of Tasks
- Precedence / Infeasiblity Relations
- Window Modification $\Longrightarrow$ New Relations

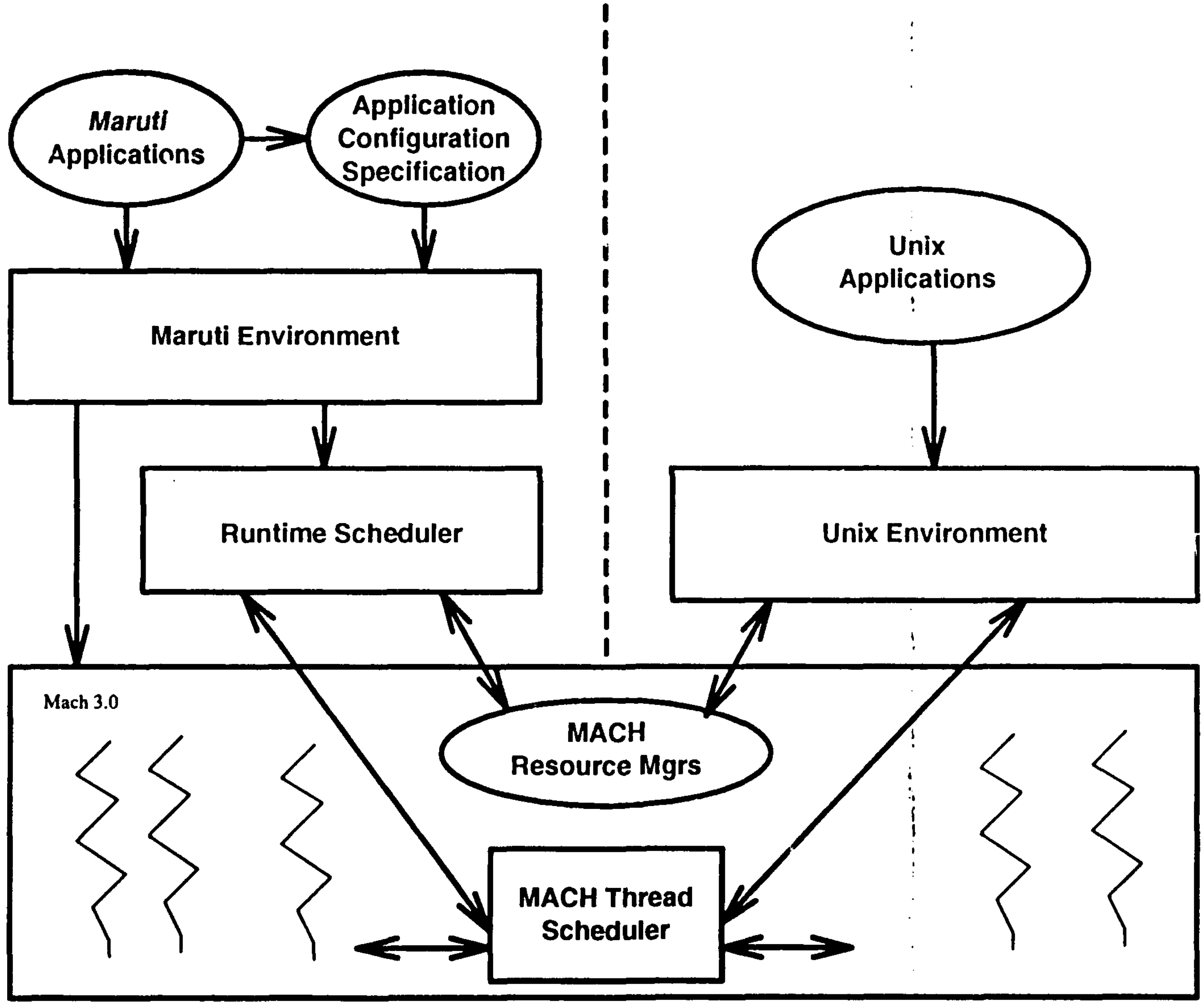
Scheduling: Analysis

- A set of precedence relations induce a partial order on the task set
- Only sequences consistent with partial order need to be considered for scheduling (reduced search space)
- Early test for infeasibility
- Polynomial Time Analysis ($O(n^3)$)
- Can Handle Precedence, Exclusion etc.
- Selective Preemption to enhance schedulability

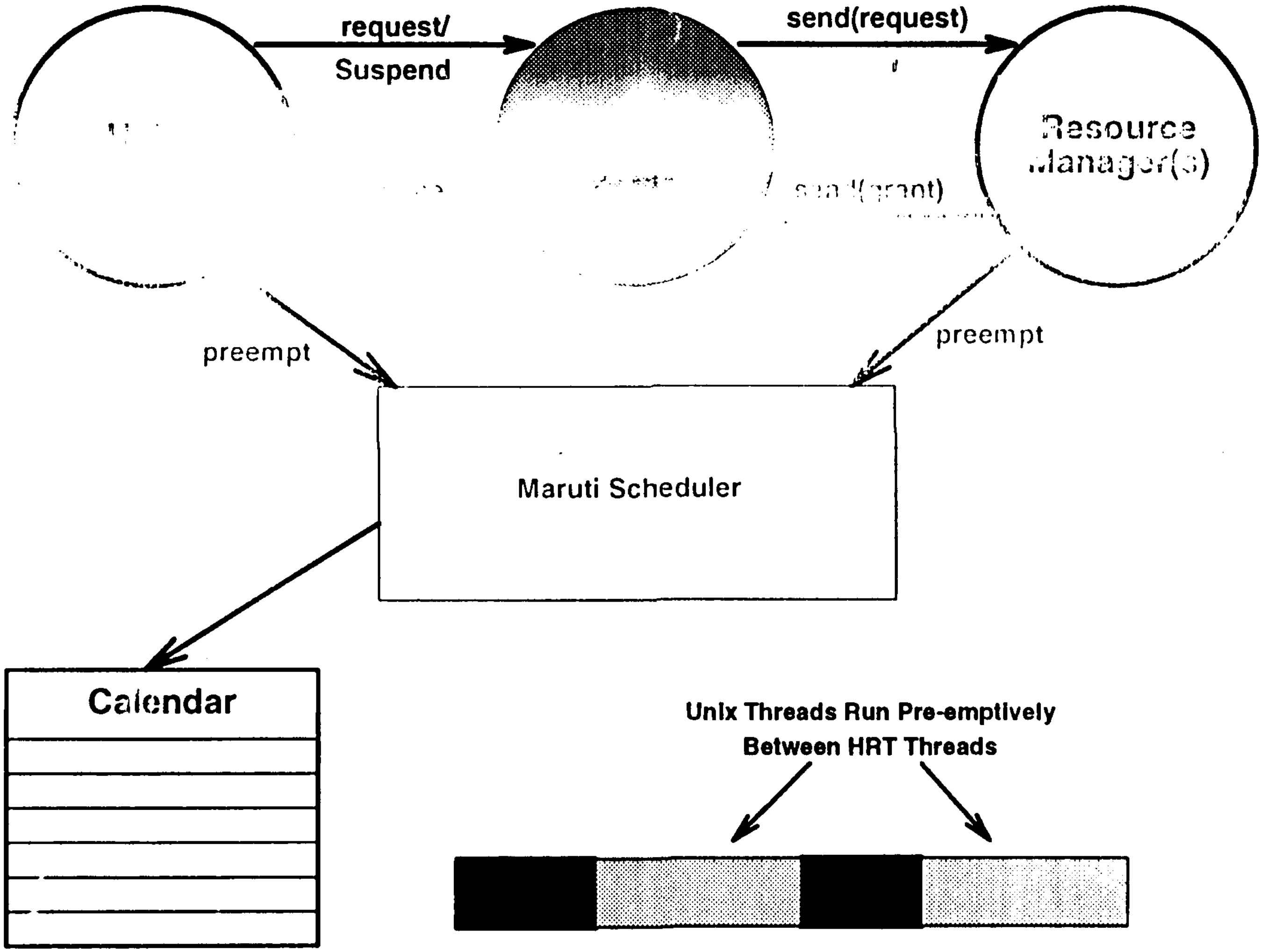
Scheduling: Mach-Maruti Interface



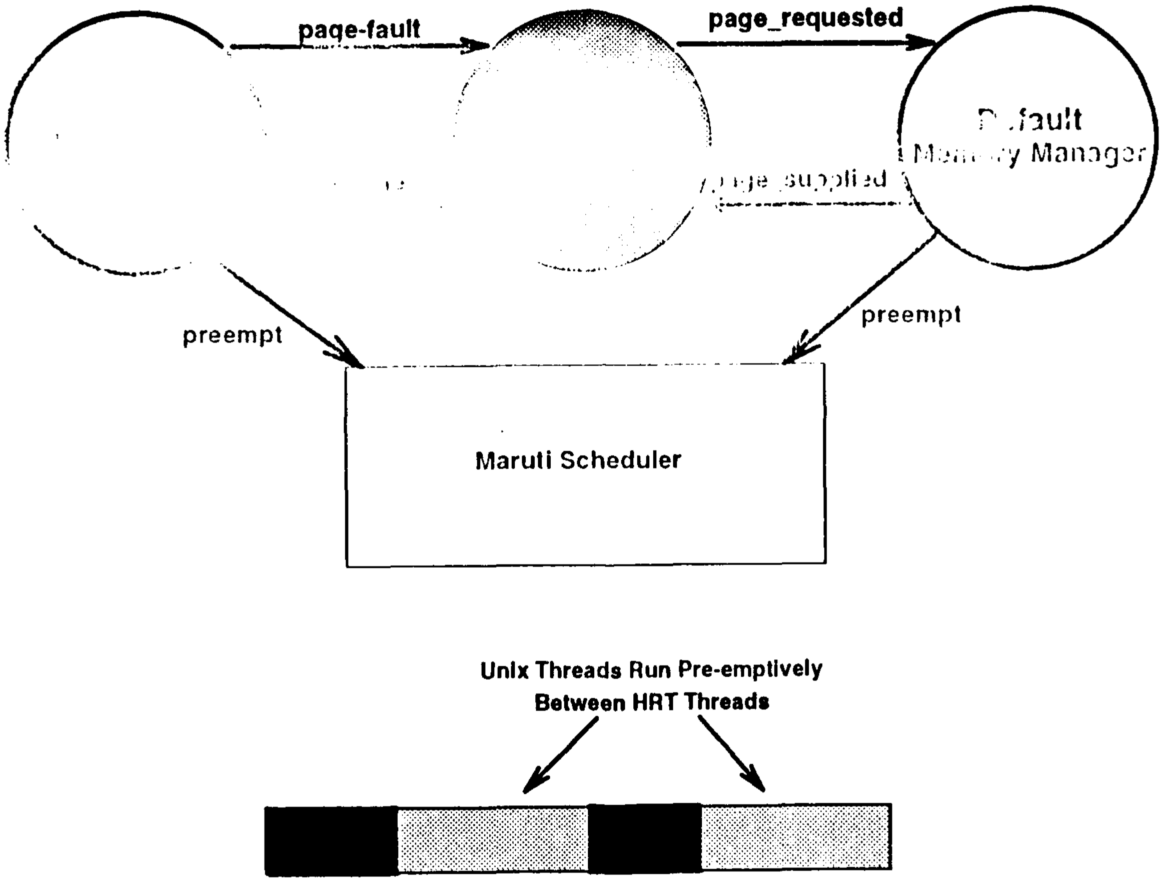
Real-Time Tasks Co-exist With Unix



Mach Enables this Flexibility



Example: Page Fault Handling



October 5, 1992

A New System Model for Industrial Plant Control

Jay S. Bayne
Bailey Controls Company
Elsag Bailey Process Automation
29801 Euclid Avenue, Wickliffe, OH 44092
216.585.5501

Abstract

The essential thesis of this paper is that the next generation of industrial-grade process plant automation and control systems requires a new system model. Key features of this new model include enhanced real-time control semantics to support the interactions of distributed fine- and coarse-grained objects. The interaction model is based on adaptive scheduling policies, dynamically adjustable system configurations, and new classes of abstract real-time data types. The emergence of distributed computing machinery to host objects based on these semantics will allow the creation of *mission-critical, vertically integrated industrial applications* whose spans of control cover a much wider operating domain of the process plant. The new system model will provide the basis for a plant control system (PCS) environment that can subsume the duties of today's more limited and proprietary regulatory distributed control systems (DCS) while providing the platform for a new generation of advanced plant controls.

Key Words: Continuous process plant control; distributed control system (DCS); plant control system (PCS); objects; threads; adaptive scheduling; dynamic configuration; real-time control.

1. Introduction

The domain of commercial industrial process control is served today by automation systems that have been optimized for *linear, deterministic, sampled-data regulatory control* problems. In the main, these systems utilize distributed microprocessor-based elements interconnected by various proprietary communications structures to implement classical analog regulatory loop control policies and mechanisms. For a given plant control application (e.g., industrial steam production) control policies are typically expressed by engineers in the semantics of feed forward and feedback control elements (e.g., the proportional integrating and differentiating, or PID, controller). These elements are driven by discrete-time, quantized measurements comprising small vectors of integer and real number values (e.g., pressure, temperature, pH, flow rate). Output from the control logic elements effect process state through actuation of final control devices (e.g., valve positioners, motor controls and electrical switch gear). This input-control-output relation defines a "control loop" and is the architectural basis for the current generation of instrumentation and control products.

More advanced regulatory control systems add to the basic mechanisms facilities for multi-loop control policies predicated on process identification, optimal estimators, optimal controllers, and high-fidelity process simulations. Some even support embedded advanced control primitives such as Smith Predictors, multi-variable and adaptive (e.g., self-tuning) controllers, and batch process controllers. These more advanced features support the creation of control policies appropriate for the management of more complex processes whose character may be less deterministic, stochastically driven, not directly observable, or only partially controllable. Services provided by these advanced functions generally define the base of what is considered the supervisory control domain.

Contemporary regulatory control systems have grown in both design and application from the bottom up, having been derived from earlier electro-mechanical, pneumatic, and analog control precursors dating back to the industrial revolution. They are typically specified and implemented by conservative operating plant personnel whose primary interests are the production of the product(s) the plant exists to manufacture. The automation systems have historically been seen as stand-alone systems whose inputs and outputs are distinct from those of other plant processes.

For example, it is not uncommon to find in a pulp and paper mill [Smoot89] three different control systems, one responsible for the power house (electricity and steam processes), one controlling the pulp mill (fiber and effluent process), and one controlling each paper machine (paper production processes) in the mill. These various control systems were probably purchased at different times, from different vendors, at different points in the evolution of distributed control technology, by different plant management personnel, for different economic reasons, and without the benefit of an overall plant integration and automation policy. Rationalization, integration, training, maintenance, and inter-operation are generally only afterthoughts, and today represent significant elements in the total operating cost equation for the plant.

At the same time individual control systems were being implemented at the operating plant level, new levels of automation addressing a different class of problems (with completely different semantics) were being applied to enterprise business systems. Business systems are typically not viewed as control systems per se. They are generally sponsored by corporate finance and MIS organizations whose problem domains are semantically different from that of plant operations, and whose policies and mechanisms have grown from a distinctly unique tradition.

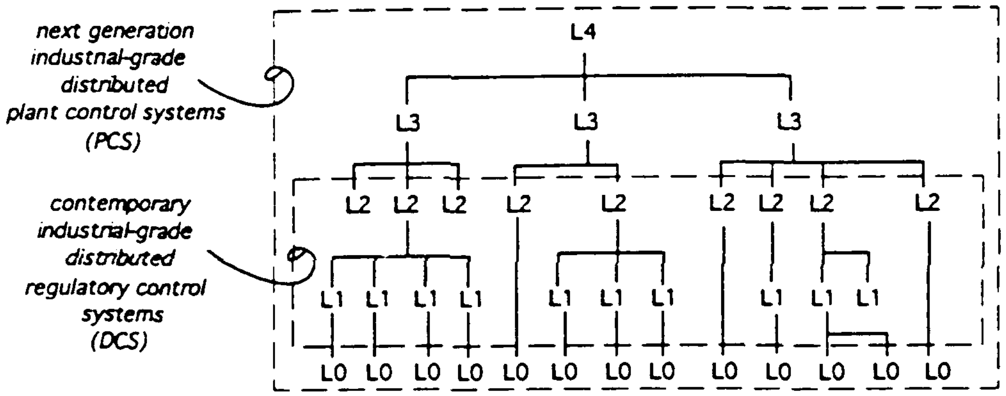
During the period from 1960 to 1980, while much of today's plant and business automation was being installed, a great deal of work was done to bring these two disciplines together. Most of this work was academic with its foundation based on principles of operations research, econometrics, cybernetics, and the modeling and simulation of large-scale dynamic systems. During the last two decades, computer science has given us a rich set of domain neutral semantics within which to express control and automation problems, solutions, policies, and mechanisms that are applicable to the traditional regulatory and business domains. During the same period global commercial pressures have made production efficiencies, product quality, and environmental and resource management issues critically important business policy and capital investment drivers.

These factors have each led to increasingly richer requirements to interconnect plant process control systems with operational business systems to facilitate such applications as optimal plant production scheduling, raw material resource planning, and compliance to regulatory agency tracking and reporting requirements. For the last decade these applications requirements have led increasingly to *connectivity* and *inter-operability* requirements that have helped create an entire industry based on the professional practice of *systems integration.* They have also given impetus to an entire spectrum of industrial, national, and international *open systems standards* movements.

It is our thesis that during the next decade world-wide commercial forces will justify the fusion of business and regulatory control policies. This will result in the rationalization of intra- and inter-plant operating policies, resulting in the establishment of consistent operating and control semantics. This unified set of operating and control semantics will foster reusable applications, encourage integration, and lower the overall installed and operating costs per automation function. The science of distributed information systems is sufficiently rich today (with a few important exceptions) to provide the essential computing and communicatio. fabric on which semantically consistent mechanisms can execute. We will explore this issue in the following pages.

2. Contemporary Industrial Process Controls

Contemporary industrial digital control systems (DCS's) provide direct digital data acquisition and control of industrial continuous, batch and discrete manufacturing processes. For a number of logical and historical reasons, these control systems fall within an automation hierarchy. The figure below depicts this logical hierarchy of the *automation platform*. The levels imply the span of influence of the control applications that populate physical components that comprise the total plant automation system. The five level hierarchy combines control elements that span field instrumentation elements at the base, to inter-plant control elements at the top.


next generation industrial-grade distributed plant control systems (PCS)
contemporary industrial-grade distributed regulatory control systems (DCS)
L4
L3
L3
L3
L2 L2 L2 L2 L2 L2 L2 L2 L2
L1 L1 L1 L1 L1 L1 L1 L1 L1 L1
L0 L0 L0 L0 L0 L0 L0 L0 L0 L0 L0 L0 L0

Level 0 L0 defines a domain that encapsulates applications (e.g., transducer management) responsible for field process measurement, actuation or analysis. L0 objects are end-systems on field communications links responsibie for either input to, or output from, a L1 regulatory or cell control. Therefore, L0 elements are generally grouped and associated with specific Level 1 control policies.

Level 1 L1 objects implement policies governing data acquisition, filtering, and regulatory or sequence control functions. L1 objects are responsible for basic manufacturing cell-level (inter-transducer) direct digital control, implementing the automation responsible for primary physical process supervision and safety-related process management. L1 has its roots in electro-mechanical, pneumatic and analog controls which provides the semantic framework for control policies and mechanisms found at this level. This level exhibits the most stringent real-time and fault-tolerant requirements within the hierarchy.

Level 2 L2 defines the supervisory or area control domain responsible for basic area (inter-cell) production control. L2 is generally associated with plant control rooms where many regulatory loops are consolidated into higher level process control. L2 objects provided mechanisms which implement policies governing operator interfaces, process data archives, trend analysis, alarm management, diagnostics, plant area start-up and shutdown, and L1 configuration and set-point control. This level generally defines that lowest level for horizontal (wide-span) control policies.

Level 3 L3 represents the intra-plant (inter-area) control domain where policies governing plantwide coordination, cooperation and control are implemented. L3 objects provide mechanisms that govern plant production scheduling, energy and raw material utilization, inventory and work-in-process, product quality, and maintenance management policies. This level provides the intra-plant control room domain, where plant management personnel can interact with the operating conditions of the overall plant. This is a site-specific wide-span control domain.

Level 4 L4 is the inter-plant (intra-enterprise) control domain providing objects responsible for enforcing global prɔduct production coordination, supervision and control. This domain is relevant to enterprises operating multiple plants with common or shared production facilities. For example, two chemical plants responsible for manufacturing a specific polymer may be coordinated to meet volume commitments under the uncertainties of maintenance, labor, transportation, and raw material availability.

From an implementation perspective, existing DCS designs are focused on two principle areas: I/O front-ends and L1 controllers. This has focused attention on the interconnection networks running between these two elements, and competitive systems today have taken different approaches to implementing their respective control networks (aka, "data highways.") The designs are optimized for low cost per I/O point (analog and digital), 10ms control loop response times, high availability, and low MTTR.

User interfaces are today, for the most part, based on proprietary graphical presentation systems. Many are hosted on general purpose workstation-class machines from HP, DEC, and Sun. The principle application engineering tools used are graphical control block programming environments that allow the process control engineer to cut-and-paste control software objects (e.g., a function generator or PID block) into a design. These graphical descriptions, each of which typically define a loop control policy, are then compiled into executable "segments" that are loaded into one or more of the L1 controllers.

L1 controllers are specially designed single board computers that are optimized for reliability, environmental hardness, n:1 or 1:1 redundancy, high-speed communications with peer L1 machines, hot insertion into their backplanes, and multiplexed communications with the L0 input/output subsystems. L1 and L0 devices generally operate in a master-slave relation, with a single L1 master responsible for up to 32 L0 I/O slaves.

For the purposes of this paper, the salient feature of L1 (and some L0) elements is the manner in which loop segments (and complex measurement, actuation and analysis) code is scheduled and executed in "real-time." Contemporary DCS implementations have taken a very conservative approach, based on best practices of the late 1970's and early 1980's when these systems were architected. There were a number of silicon-based executives available such as VRTX, MTOS, and pSOS that provided the basic multi-tasking kernel primitives required for interrupt-driven, priority-based scheduling. These kernels were optimized (for the time) for low task switching overhead, priority-based queuing and task dispatching, and primitive interprocess signaling. To simplify the designs, and to keep context-switch latencies to a minimum, DCS vendors resorted to the simplest of all task scheduling policies -- fixed priority within fixed time cycles.

The various loop control mechanisms, as expressed in sequences of compiled and linked control block segments, are deposited by a systems engineering development tool, typically hosted on a PC- or workstation-class machine into the address spaces of L1 machines. The loading policy is based on a priori knowledge of typical execution profiles of various control blocks. The configuration tools estimate the load a given loop control policy will likely place on the L1 machine, its periodic execution requirements, and its estimated duty cycle, or completion time. On the basis of on these factors (and a few heuristics-based *magic numbers* thrown in for good measure) a given L1 machine is assigned its task set.

Task sets and their interaction profiles are rarely understood well enough at design time to guarantee their correct temporal behavior. To compensate for this, L1 machines are typically under utilized in terms of processor cycles. Furthermore, a great deal of verification testing and system tuning is performed during the factory acceptance and on-site system commissioning phases of a project. This multi-step configuration process is required for establishing confidence that the system logic and its implementation are, in some fashion, *correct*. This process is not only time consuming and cumbersome, but represents a significant cost, both before commissioning and afterwards during maintenance, upgrades, and redesign caused by process changes within the plant.

This effort at scheduling L1 task sets is at the core of the DCS configuration problem. It does not address the scheduling of L2 or L3 or L4 tasks, nor does it manage the interdependencies among tasks at the various levels in the hierarchy. Therefore, it can be said that although L0 and L1 are truly "hard" real-time, as defined by [Cheng88] and others, L2-L4 are only "soft" real-time. Therefore, applications that are vertical in nature (i.e., engage the resources of L0-L4 machines on behalf of some computation) are only soft real-time, at best. We require the next generation of industrial control systems to be "vertically hard" real-time, as opposed to today's machines that are "horizontally hard," and only so at the lowest levels.

Because current DCS configurations are optimized for (and typically procured and applied to) L1 and L2 applications, their resources are highly utilized at L0 and L1, but often under utilized at L2. There are a number of reasons for this disparity. They are primarily historical, but certainly the proprietary nature of contemporary DCS implementations makes it difficult and expensive to realize integrated vertical solutions to advanced plant control problems. This situation has lead to the development of "middleware" companies such as Oil Systems and Setpoint that have successfully developed and deployed a limited number of control-oriented applications on standard general purpose computers that bridge the gap between "business systems" at the top of the hierarchy and the "control systems" (DCS's and PLC's) at the bottom. These middleware services tend to support market- and process-specific applications requiring real-time archival storage, analysis, and presentation functions.

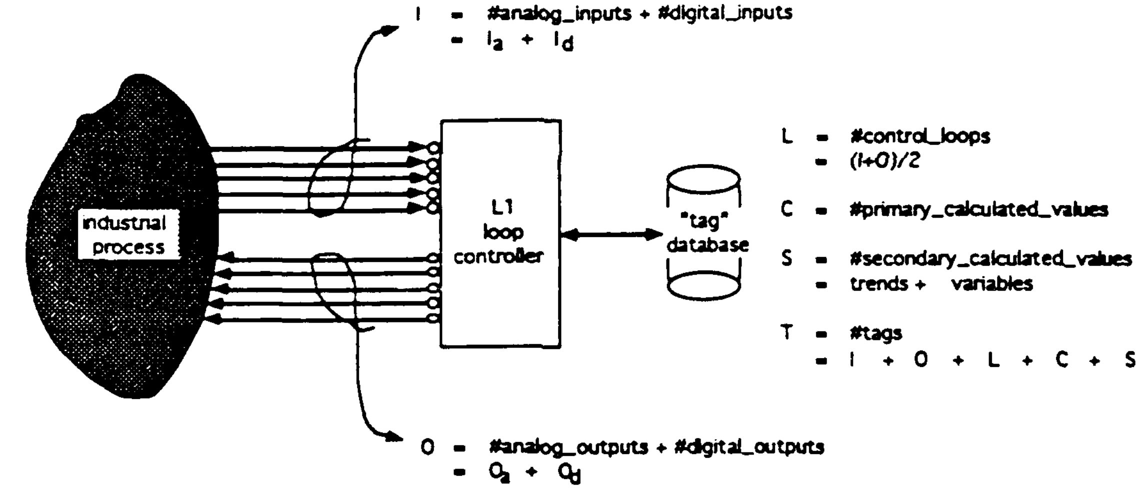
The historical reasons for this control domain isolation and under-utilization are rooted in the business practices of both vendors and end-users of regulatory control systems. It is difficult for vendors to think outside their historical context and base market applications. For example, Bailey is rooted in the electric utility and industrial steam markets, Honeywell is rooted in the petroleum refining market, and Allen-Bradley is rooted in the electrical switch gear commodity market. Furthermore, the problems of control at L1 are complex, and industry-specific process knowledge is an asset that must be developed and nurtured over time.

On the end-user side of the equation, there are a number of impediments to developing vertically integrated control policies and mechanisms. First, L3 and L4 are not well understood as control domains. They are still today referred to in MIS, finance, or manufacturing terms. Practitioners at these levels do not think in terms of real-time control. Their semantic frames of reference are rooted in transaction processing, database management, COBOL, MRP, and Lotus 123. To make matters worse, sponsors of L3 and L4 automation initiatives generally do not define their problem domain in terms relevant to process plant operations personnel.

The semantic gap between vendor and user, and among user communities, will likely persist for some time. It is a condition of current business practices and tradition. The next generation of plant control systems need not be so constrained. There are cogent reasons to believe that providing control platforms and associated services designed to support the construction of vertically integrated applications will be a primary driver for resolving (dissolving) this disparity.

2.1 Contemporary Platforms

Although there are a number of important distinctions among the competing distributed control systems in the market today, they are all much more similar than they are different. One set of metrics relates to the size and complexity of the control tasks the systems are commissioned to manage. Typical control system projects can be measured in terms of I/O counts, control loops, and system database elements (aka, "tags.")



The figure above provides a simplified model of a node in a contemporary process control system. The graphic depicts a four loop controller and its associated local database. A "control loop" is defined as an input/output pair wrapped around some control algorithm. In many supervisory systems, the primary function is data acquisition (SCADA) for which this I/O pair-per-loop rule is violated. Such systems may have 10-20 inputs for a single output. Inputs comprise analog and digital measurements, while outputs are either analog or digital control signals. During the course of computing (estimates of) process state changes and requisite control policies, primary and secondary derived quantities and intermediate calculations are made. All of these elements combine to define the contents of the "tag database" associated with a given controller.

A large system configuration might, for example, contain a distributed database of 30,000 tags derived from 10,000 L0 I/O points, of which 1000 pairs are associated with 1000 L1 regulatory control loops. There may be another 7000 independent L0 process variable measurements that provide input to 100 L2 supervisory controls responsible for producing 900 outputs.

From these points and L1 loops and L2 controls are derived the 12,000 primary process *meta-variables* (e.g., averages, process state estimates, etc.) and another 8,000 secondary metrics (e.g., trends, correlations, etc.) The 1000 L1 loops might be implemented in 50 controllers (hardened, single-board computers) hosting an average of 20 control loops each, some of which are dual-redundant for safety reasons. The 100 L2 controls may be implemented in 10 L2 controllers, some of which are redundant. These 60+ controllers would be connected to their respective input-output channels through collocated I/O subsystems. Each would maintain the real-time status of their tags. These tags would be available throughout the control system by virtue of the system's distributed database and communications services.

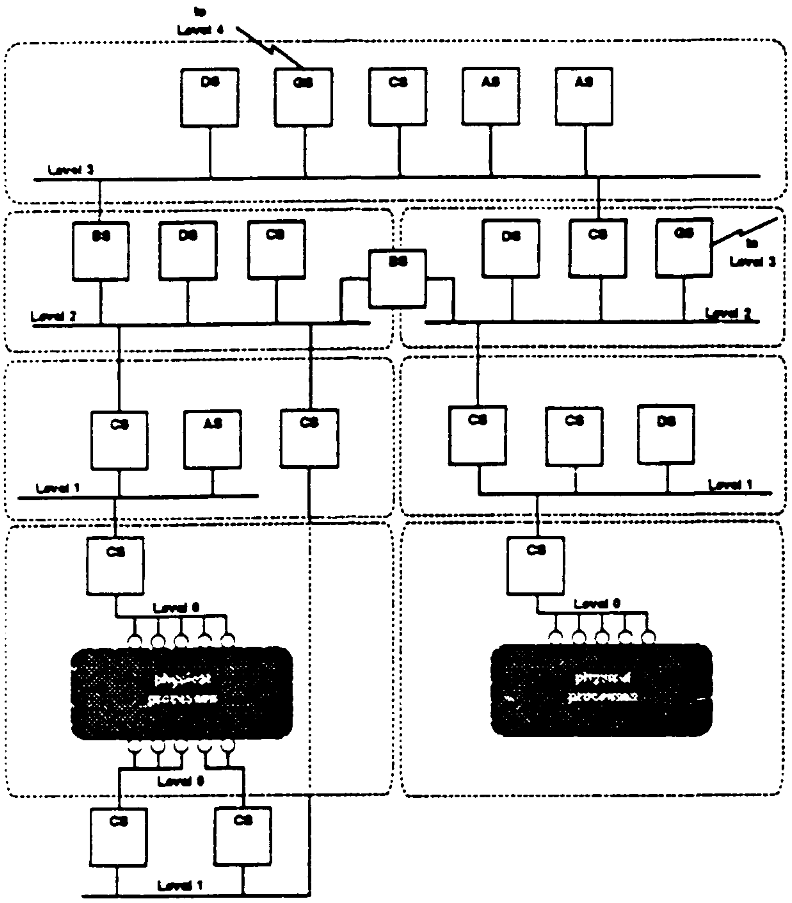
Systems of this size are rare today (<10% of the total), but occur frequently enough that system designs must take them into account. Many of today's DCS vendors sell systems that cannot be easily scaled either up to this size, or if designed for this size, cannot be scaled down to small economically viable systems. Scalability is a critical design requirement for the next generation of commercial control systems.

The figure below depicts a platform configuration that is typical of today's plant automation systems. L0 field devices are connected through some form of a field bus to L1 local or remote I/O processors or integrated controllers. L1 devices are interconnected via control network segments to L2 area controllers that are, in turn, connected through a backbone network to general purpose L3 computing devices (e.g., via Ethernet-based LANs). These L3 devices are then used to interconnect to enterprise-wide business systems (e.g., through IBM SNA-based networks).

This general configuration associates control domains with communications sub networks, since the physical geometry of the plant most often defines both the physical and logical partitioning of the process control problem. In the next generation of control systems there are clear cost and performance incentives to collapse the three physical networks (Level 1-3 in the figure above) into a single high-performance structure. There will, however, remain practical reasons to partition the control problem into a logical hierarchy that is appropriately mapped onto the flattened physical structure.

The figure shows a number of different processing nodes representing "control domain hosts," or servers. In this simplified example, control servers (CS) interact with transducers directly interfaced to plant processes. Applications servers (AS) interact with the control services to produce derived process state information. Display servers (DS) provide the domain specific operator interfaces which can be shared across the system at end-user devices (in the sense of X-windows). Bridge servers (BS) isolate control domains and route, through flow control protocols, high priority safety-related process synchronization messages. Gateway servers (GS) provide for interconnecting subsystems into heterogeneous internetworks.

As logical as this picture is, contemporary systems have not solved in a unified manner a number of key problems, including issues related to L1-L2-L3 interoperation, CS-AS cooperation at a given level or across levels, GS functionality as it pertains to access control and accounting in an internetwork, or the style and function of shared displays (i.e., the "single window" operator console). These problems are difficult (if not intractable) given the heterogeneous, loosely coupled, and proprietary nature of current control system designs. The situation is exacerbated by the conservative policies governing control of



processes with safety-related side effects and the nature of capital project justification and implementation cycles within end-user markets.

Even without a "unification principle," the process industries have succeeded in building systems in an ad hoc fashion out of multi-vendor components with reliance on in-house and third-party systems integration services. The predominant trend is to utilize combinations of PC-, workstation-, and VAX-class machines interconnected by LAN's and LAN-servers for the L3 and L4 application platforms. These are in turn attached through non-standard means to more homogeneous L1-L2 DCS or PLC environments. These configurations tend to be relatively inexpensive from a hardware view point. However, the less tangible costs associated with integration services, licensed software, system maintenance and development, documentation, and training tend to be very high when compared with more homogeneous solutions. The goal of next generation plant control systems is to provide greater benefits and lower total costs through rationalizing these ad hoc control platforms.

3. Next Generation Industrial Plant Controls

The next generation of industrial plant control systems will be architected to provide new capabilities while at the same time addressing the deficiencies of the current generation DCS+LAN products. New capabilities will likely include integrated vertical applications, wider (aka, plantwide) spans of control, greater process fidelity, improved availability of the whole ensemble, lower total costs per control function, and backward compatibility to legacy systems.

The half-life of any new control system is estimated at 10 years, requiring its design basis to support an evolutionary and adaptable implementation path. The design of a new contro machine is predicated on a number of base hardware technology enablers, certain system anc applications software paradigms, and competitive and technical pressures. For the purposes of this paper, we will not consider market or financial drivers, although they are in many respects more important than technical issues.

3.1 Hardware Technology Drivers

There are a number of important hardware trends that must be considered. Circuit densities are increasing at about 25% per year, doubling every three years [Hennessy90]. Device speeds are increasing at a similar rate. This is equivalent to realizing the same device functionality in half the space at twice the speed every three years. As a related development, the cost per processor instruction cycle is declining at 25% per year. This yields 100% additional processing capacity (operating at twice the speed in half the space) for the same cost every three years. The basis today is 25 MHz machines. By the mid-life of a new system we will be able to use 200 MHz processors in the same physical space and at the same prices as today's machines.

The cost of memory is declining at 15% per year, dropping by a factor of twc every five years. DRAM densities are increasing at about 60% per year, quadrupling every three years. Therefore, in the span of just 10 years we should see twelve times the memory density at one quarter the cost. At the same time, application address space is being consumed at one additional address bit per year, on average, suggesting we need an additional 10 bits of address over the design half-life of a new machine. In today's control systems we use about 17 bits of address space per L0 device, 21 bits per L1 device, 23 bits per L2 device, and 24 bits per L3 device. By the year 2005 we estimate that L0 devices will utilize 26 address bits, with 32 bits at L1, 34 bits at L2, and 36 bits at L3. Clearly, 64-bit processors are required to implement the upper domains of the next generation of machines.

Disk density is increasing about 25% per year, doubling in three years [Hennessy90]. This keeps pace with the consumption of DRAM, and suggests that over the life of the system secondary storage demands will increase for two principle reasons. First, backing storage is required to contain (at least part of) the static images of the L1-L4 machinery. Second, significant archival storage is required to log the operating history of the plant. For example, a plant with 1,000 field measurements sampled at 1 Hz would produce a raw L0 data rate of 64 Kbps, assuming 64 bits per point (data, plus status, plus time stamp). That represents a potential uncompressed L0 storage requirement of over 2 Terabits per year, or 250 Mbytes per point per year. Assuming an average compression factor of .6, we can estimate an appetite of 150 Mbytes per point per year of required archival storage capacity.

Available communications bandwidth is increasing by a factor of 10 every three years. Its basis today is 10 Mbps, yielding 100 Mbps by 1995, and 10 Gbps by 2005. This bandwidth is expected to be absorbed for a number of reasons, primarily at automation levels L2 and L3, including: i) the routine use of multimedia man-machine interfaces that support integrated voice and full-frame video display systems; and ii) the increasing utilization of optical sensors. These sensors have application in many control domains, but when used for high speed flat sheet production (such as steel, film and paper making) can produce enormous volumes of data in very short periods.

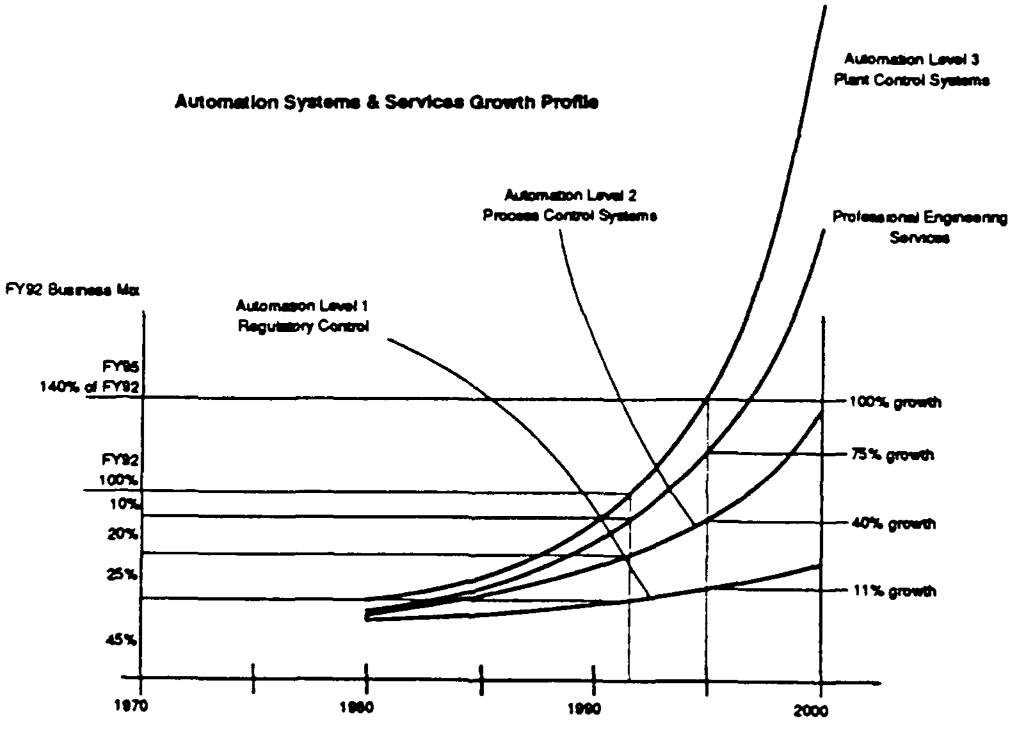
This brief summary suggests that by the end of the design half-life of the next generation of plant control systems (circa 2005) the computational nodes of the system will routinely be operating at 200 MHz, supporting an address space of 30-40 bits, intercommunicating at

10 Gigabits per second over an optical mesh, collectively tracking and controlling an evolving plant state comprising over 10^6 objects, and utilizing Terabyte backing storage subsystems. This scenario points to the real design problem -- software -- its creation, configuration, deployment and maintenance.

3.2 Software Paradigm Shifts

Building high capacity real-time distributed computing systems has been motivated in a number of applications domains, including military command and control, industrial process control, public transportation, and telecommunications applications. These domains are closely related in terms of growth in demand for real-time distributed hardware and software based functionality. This demand in the industrial automation market segment is characterized in the figure below.

In FY92 it is estimated that 45% of revenue derives from good old fashion L0-L1 regulatory controls. 25% derives from more advanced L2 supervisory controls; 20% derives from professional engineering services across the domains, but not including custom applications software development and systems integration; and 10% derives from L3 plantwide control functionality. Over the four year period ending in FY96 it is estimated that the process automation industry will realize an 11% CAGR in its core Level 1 business, 40% CAGR from Level 2 supervisory control, 75% CAGR from professional services, and 100% CAGR in the Level 3 plantwide automation sector.



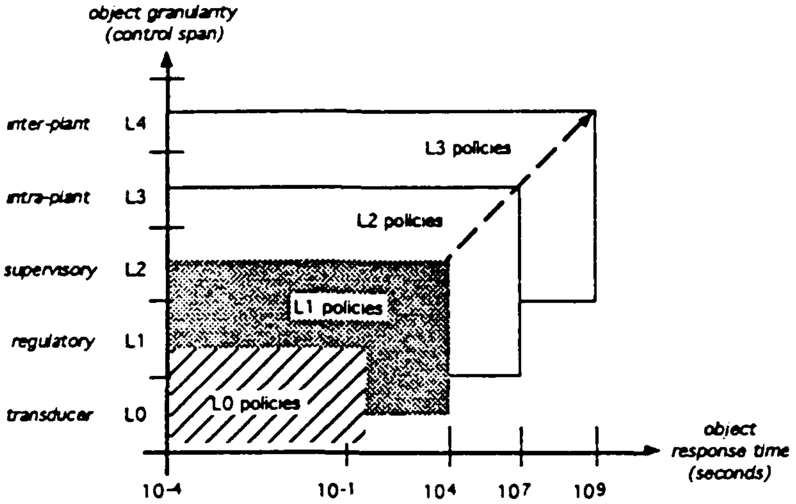
The absolute numbers are debatable, but the trends are clear. Over the next five to ten years the automation market is expected to grow by well over 50% CAGR in the consumption of upper level and inter-domain applications of control. Within the L1 control domain it is expected to grow less than 15%. Any strategic investment in technology must clearly support the development of Level 2 and Level 3 control software, its vertical integration, and its attendant professional services products.

The definition and implementation of software capable of *correctly* controlling a plant that is hosted on a distributed computing system provides a significant technical and marketing challenge. Classical programming models, methodologies, and tools are not sufficiently rich to handle either the complexity or volume of *validated code* that can be hosted on distributed multi-processor configurations whose nodes routinely support 40 bits of addressability. A new programming model is required, one especially focused on heterogeneous multiprocessing of mission critical applications.

The work we and others have done supports adoption of a virtual machine model based on *objects, ports, and threads* as demonstrated under various assumptions in the Mach [Rashid86] and Alpha [Northcutt87] micro-kernel projects at CMU, Chorus from Chorus Systemes, S.A. [Rosier92], OSF/1 from the Open Software Foundation [Leopere92] [OSF92], and the Mach/RT project at the Center for High Performance Computing (CHPC), Worchester Polytechnic Institute [Shipman92]. We will return to the implications and application of this programming model in a later section, but will first consider the real-time control problem domain from an applications level perspective.

3.2.1 Integrated Vertical Applications

The figure below defines the control problem space for the next generation of control systems in terms of object granularity (control span) and response time (related to state variable persistence and control fidelity). Industrial automation systems must be capable of hosting applications whose access rights and essential resources extend from L4 downward through L0. Guaranteed response times must have lower bounds in the sub-millisecond range. Measurement and derived process-state must be stored for as long as decades. The business opportunity expressed here is to provide platforms and related applications that follow the arrow up the commercial "food chain" (as depicted in the previous graph). As hardware platforms and system operating software are continually rationalized (i.e., standardized and made commodities), the real value added to the marketplace will be i) the L0-L1 front ends, ii) the horizontal and vertical control applications, and iii) the attendant professional services.



To realize systems within this space that adhere to "hard real-time" [Jensen92] operating constraints we require a new approach to system design. The development of control policies and mechanisms that engage the services of objects arrayed vertically from L0 to L4 requires a programming model that is fundamentally different from that employed in contemporary systems. First, the model must provide semantics that are consistent across the levels. Second, the underlying hardware environment must have known performance and reliability measures. Third, the system must be scalable, since plant control policies and mechanisms, and end-user requirements will evolve over time. And fourth, the next generation of automation and control systems will have to connect to, and interoperate with, the thousands of installed legacy systems already in place world-wide.

"Vertical application" refers here to the capability to engage the services of objects within the sensor-actuator, regulatory, supervisory, and plantwide domains on behalf of specific plant control policies. This requires defining mechanisms in a semantically consistent manner across the domains. *Continuous emissions monitoring* is a good example of a commercially relevant inter-domain application. Applications of this type require the processing of a variety of data types with wide variations in temporal granularity, the preservation of measurement profiles on long-term stable storage, often complex man-machine graphic presentations, and relatively complex numerical methods.

In heterogeneous systems, vertical applications require adherence to inter-level programming interfaces up and down, and across the hierarchy. POSIX [Zlotnick91] and DCE [OSF92] are examples of well-defined interfaces pertaining to the boundary layer between applications and operating system software and among applications. Such interfaces must be defined for the application domain as well (i.e., distributed control system semantics). The interfaces may or may not be standardized a la ISO, but for the application to be stable over the life of its implementation, the interfaces must at least be stable. In a homogeneous computing environment the inter- and cross-level interfaces will be stable by definition.

Representative applications identified for deployment within the intra-plant control domain vary by industry and business practice. The figure below gives some flavor for their character. These "multi-vendor applications" are typically bolted together to satisfy the needs of customer sales-order projects. The process is classical systems integration work, generally at or above L2, resulting in ad hoc solutions with limited reuse potential. Furthermore, the hardware and software elements above L2 are typically outside purchased equipment and under the design control of third-parties. Therefore, the total operating cost of the resultant system can be rather high given the maintenance and complexity of upgrading such systems.

Level 3: **Plantwide**	Total Quality Control Prod Inventory Mgmt Distribution Mgmt Plant Financial Mgmt Product Quality Mgmt Personnel Mgmt L2 Synchronization	Maintenance Mgmt Purchasing Order Processing Safety & Security Environmental Systems Production Scheduling Plant Utilities Mgmt	Laboratory Data Mgmt Statistical Analysis Plant E-Mail/Calendar Office Document Mgmt Plant Engr Mgmt Elect Filing Systems Plant Alarming	Plant Relational DBMS Tech Document Mgmt Material Req Planning Capacity Planning Training/Simulators Health Safety Systems Communications
Level 2: **Supervisory**	Real-Time SQC Process Computations Advanced Area Controls L1 Synchronization	Area Energy Mgmt Laboratory Sampling Process Alarming Process Modeling	Control Room Mgmt Area Scheduling Process Optimization Area Alarming	Emergency Shutdown Batch Recipe Mgmt Area Safety Systems Communications
Level 1: **Regulatory**	Closed Loop Control PLC Sequencing Fault Isolation L0 Synchronization	Loop Safety Control Redundancy Scan & Data Collection Cell/Machine Control	Lab Automation Measurement Calibrate Regulator Tuning Process Alarming	Process Modeling Actuator Calibrate Adaptive Control Communications
Level 0: **Transducer**	Sensors Actuators Signal Conditioning Synchronization	Robotics Smart Sensors Smart Actuators Drives	Self Calibrating Diagnostics Sensor Alarming Actuator Alarming	Signal Filtering Signal Estimation Communications

An important goal of the next generation of control systems is to "objectify" these applications through the definition and use of standard interfaces at the application (e.g., DCE) and operating system (e.g., Mach) levels. This should result in moving the problem of constructing integrated control applications from a programming problem to a software configuration problem, with a concomitant reduction in total-installed and total-operating costs, an increase in reusability, and an improvement in total product quality and customer satisfaction.

3.2.2 Wider Spans of Control

Applications with wide control spans (e.g., those at L3) reach across many lower level, yet autonomous, control domains. These applications invoke the services of objects at levels above and below the end-user domain, but the execution threads spend most of their time wandering the object domains at lower levels. The semantics are a mixture of client-server and peer-to-peer, neither strictly dominant. Energy management within a multi-area plant (e.g., a pulp and paper mill) is a good example of a wide-span control problem with cooperating process semantics, but also containing global optimization, or client-server relations. Energy management is considered a L3 function (by the nature of its end-user's management position) that requires services from many essentially independent L2 and L1 control objects.

3.2.3 Greater Control Fidelity

Higher performance systems are possible by the nature of the technology-cost curves we are riding. Higher performance begets greater speed, higher precision, and increased functionality leading to greater control fidelity. Increased fidelity implies higher loop-level bandwidth, increased control accuracy, and greater persistence of abstract data types utilized within the system. Larger, more reliable storage allows operating history to be brought to bare on real-time control strategies, facilitating the construction of expert systems and learning automaton. These opportunities are clearly control level-dependent.

At L0 control fidelity refers to the precision, in both time and value, of the input and output transducers. At L1 fidelity concerns the accuracy of the process models that govern regulatory control policies. L2 fidelity concerns the correct synchronization of regulatory policies on behalf of supervisory control functions. And L3 fidelity, albeit more coarsely grained, provides precision to computations governing supervisory domain scheduling and associated plant coordination and control functions. Fidelity in these various control domains is also application specific. The next generation of control systems should therefore provide for the specification and implementation of policies and mechanisms governing *domain-specific fidelity*.

3.2.4 100% Availability

Availability is at the center of what is meant by mission-critical control. The next generation of control systems must be inherently survivable under a broad range of system failure modes and plant operating conditions. At the heart of the survivability issue are requirements for fault-isolation, graceful degradation, dynamic reallocation of resources, and task migration semantics. Alpha micro-kernel semantics [Clark92] are, for example, particularly well suited to provide the underlying distributed operating system mechanisms for implementing high availability.

Availability has its roots in hardware design. The platform must be constructed to support redundancy of processing nodes, communications paths, and operator interfaces. The critical factor is, once again, robust system and applications software.

There are three levels of software that are of concern. At any given node in the control network there will exist components of a distributed application domain-independent real-time operating system (D/OS). The D/OS provides a reliable and consistent distributed system programming model, or virtual machine, to the next layer of control node software.

This next layer, or real-time middleware, comprises control-domain specific services that create on top of the general D/OS virtual machine services the *personality* of a homogeneous, fault-tolerant, high-availability plant control system. This personality defines a consistent model of a plant control system (PCS) that is supported by specific hardware features which are required, but not necessarily seen, by the control applications that are at the next abstract layer.

The top layer of software is seen by the PCS programmers who are tasked with creating solutions to customer control problems. This layer of programming abstraction is optimized for the problem-domain, and presents an environment where availability can be assumed (i.e., provided by the underlying layers) and abstracted out of the L0-L4 domain. Thus the D/OS and the PCS layers are responsible for providing services that guarantee the reliable operation of the control platform, and the application programming layer is responsible for providing services that control the plant.

3.5 Lower Per-Function Cost

The cost structure of contemporary control systems, as perceived by the end-user, is a critical design factor that today is centered on the idea of "cost-per-point." This bias has its roots in the low level sensor-actuator-regulator manufacturing roots of DCS and PLC vendors, and the resulting conditioning of the market. For example, analog input or output costs about $50/point today, or $800 for a 16-point multiplexed A-to-D card. On the basis of the trends discussed in Section 2.1, a 30,000 tag system containing 10,000 I/O points would cost $.5M for the L1 front-ends, not including the L0 devices. The next generation must treat this costing as an initial ceiling, and follow the relevant cost-performance-volume curves.

In the next generation of control systems software will be the single largest cost to produce and maintain, so packaging and pricing strategies will have to be significantly altered from those used today. In addition, since the functionality offered at L1-L3 represents layered software services (e.g., D/OS -> PCS -> specific customer control applications) there are opportunities for "packaging control" in new and creative ways, perhaps bundling it with its requisite I/O, and offering the ensemble as a formalized subsystem. These issues are open, yet relevant to the development of the next generation of control systems, for they establish the cost-per-function profiles which govern the design and implementation decisions.

3.6 Backward Compatibility

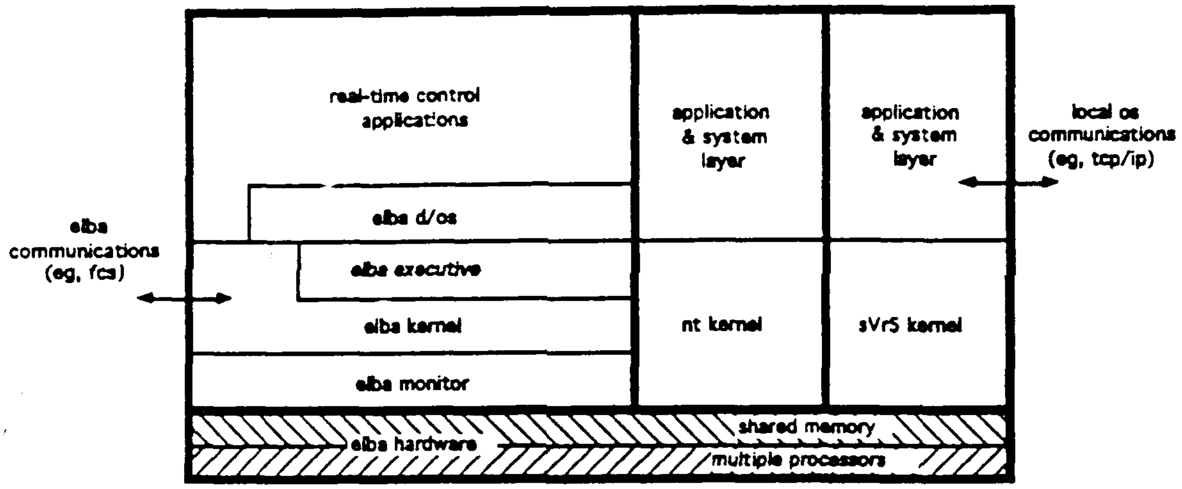
Backward compatibility is an absolute requirement. The continuous process industries build and operate plants that exhibit 10-15 year half-lives. The automation infrastructure of those plants must exhibit the same installation life-times. Therefore, the next generation of control systems must connect to and interwork with these legacy systems.

There are essentially three means to accomplish this end. The first is to faithfully emulate the legacy systems (i.e., their applications and hardware characteristics) within the new environment, and to connect to the older low level I/O systems. The second is to host only legacy applications, with certain restrictions and caveats, and ignore the faithful emulation of legacy hardware. The third is to develop a formal gateway through which the next generation system views the legacy machinery as a server. A formal legacy system service must be provided through one or more of these mechanisms, most likely a limited version of the second, and definitely the third.

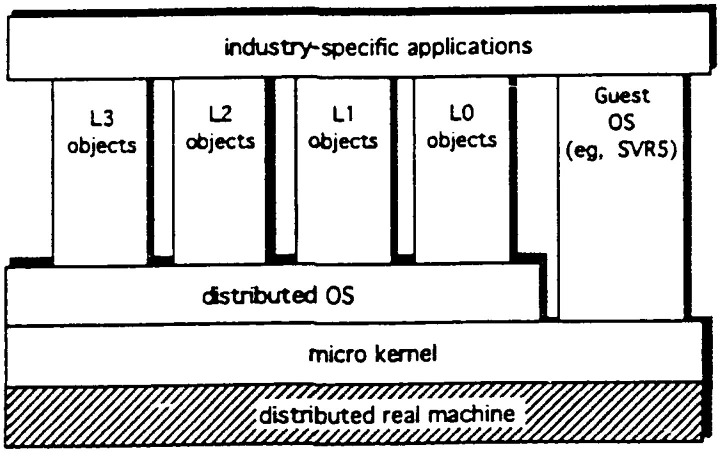
4. Programming Semantics for Integrated Controls

The semantics we propose here are independent of hardware platform. The hardware is abstracted out of the programming domain by the underlying distributed kernel and operating system services (D/OS.) The hardware environment for the PCS can be viewed as a multicomputer system comprising one or more multiprocessor nodes interconnected by a communications network that makes the multicomputer transparent to the application. The underlying D/OS provides services that expose the distributed nature of the hardware should the application require it, but a basic principle of the PCS is to mask from the applications any access to the real hardware of the machine. The stylized run-time environment of the Elsag Bailey machine node is depicted below.

elba processing node configuration


real-time control applications
application & system layer
application & system layer
local os communications (eg, tcp/ip)
elba communications (eg, fcs)
elba d/os
elba executive
elba kernel
nt kernel
sVr5 kernel
elba monitor
shared memory
elba hardware
multiple processors

The real-time application domain of the PCS is graphically depicted below. Market-specific applications, such as gas turbine controls, are defined in terms of interactions among specific control domain objects. These objects define the abstract data types that specify behaviors of control elements, according to some application-specific control policies. The various industry-specific object libraries will share many common elements, from physical plant elements (e.g., valves and motor controls) to control strategies and mechanisms. Furthermore, L_i control elements may inherit L_{i-1} object properties, defining higher level *meta-objects*. This expanding scope in the control hierarchy is depicted in the figure horizontally, right to left. The figure indicates that the application has (potential direct) access to objects residing at each control level (i.e., to objects with varying spans of control). These capabilities are restricted by the definition of the application and its activation.



4.1 Micro-kernel Level Semantics

The underlying kernel semantics are based on a fusion of the Mach 3.0 micro-kernel [Loepere92] and those features of the Alpha OS [Clark92] responsible for real-time resource management, as being implemented in Mach/RT [Shipman92]. The fundamental abstractions supported at this level are

- task..... the unit of resource allocation, a container to hold references to resources (handles) in the form of a virtual address space, a port name space (set of port rights), and set of threads
- thread..... the unit of processor utilization - an execution point of control within a task *defined by a program counter, register* set and stack
- port..... a unidirectional communications channel between tasks, accessible only via send/receive capabilities
- port set..... a set of ports that can be treated as a single unit for the purposes of receiving a message
- port right..... a capability allowing a task to exercise certain access rights to a port
- port name space ... an indexed collection of port names, each of which names a specific port right
- message.... a typed collection of data objects passed between two tasks
- message queue.... a queue of messages associated with a given port
- virtual address space .. a sparsely populated index of memory pages that may be referenced by the threads within a task
- memory object..... an internal unit of memory allocation that represents the non-resident state of the memory pages backed by this object
- memory cache object ..... a kernel object that contains the resident state of the memory objects
- processor..... a physical device (cpu) capable of executing the threads of a task
- processor set..... a set of processors, each of which can be used to execute threads assigned to the set

- node..... an individual multiprocessor within the PCS multicomputer environment
- host..... the distributed PCS platform hardware taken as a whole
- device..... a physical device (resource) available to a user-mode task, such as a L0 transducer available to a L1 task
- event..... an (asynchronous) signal dispatched by the kernel to zero or more tasks executing on the multicomputer

4.2 D/OS Level Semantics

The semantics of a distributed real-time operating system (D/OS) suitable for the next generation of plant control systems (PCS) is subject to debate, and will likely be of a proprietary nature. The control environment and its requirements for operating system services remains largely at the discretion of the implementer, since the general requirements of "open systems" do not directly apply to the domain of real-time control. This is especially true within an industrial control setting where the manufacturing processes (e.g., recipes) used to create products are themselves of considerable value. With the ability of mounting guest operating systems on selected nodes of the PCS, the requirements for openness and inter-operability are easily achieved.

Bailey Controls, in conjunction with the Elsag Bailey Process Automation Group of companies, is currently defining D/OS services specifically tuned for the requirements of industrial continuous and batch process automation. The specific details of this layer will be defined in subsequent papers, but several of the salient features can be outlined as follows.

Under specific *configuration rules*, the *kernel-D/OS environment* will form the basis of a class of virtual machines that can be scaled onto physical machines ranging from L0 devices (e.g., pressure transmitter) through L4 devices (e.g., energy management controllers). These control subsystems may then be (incrementally) blended into a larger control system ("mesh") with minimal risk. The requirement for overall system scalability demands that D/OS services are themselves partitioned in such a way that control nodes can be added and deleted from the mesh dynamically, with minimum upset to the operating plant.

The physical configuration of the PCS is determined by a control problem-domain rule base that takes into account domain complexity (control policy semantics), redundancy requirements (availability), inter-node distances (plant topology), and speed. These factors contribute to the specification of the number of nodes, their respective compute and storage capacities, the number and speed of interconnecting links, the size of the name-space within and among node multiprocessors, and the configuration of D/OS services. The aggregation, at one end of the configuration spectrum, is a uniprocessor with dedicated I/O (a small "entry level" system). At the other end, it is an n-cube mesh of multiprocessors. We believe that the practical range of PCS configurations will lie in the 1- and 2-D mesh structures that support n:1 and 1:1 redundancy among processor sets that define (at least) logical nodes dedicated to specific plant control tasks.

The mesh concept views nodes (one ore more processor sets) as execution sites for control policies that are relevant to specific control domains. Nodes can be associated into node_sets that host the regulatory and supervisory applications. Nodes and node_sets generally show affinity by the nature of the control policies that execute on them, and by the resource scheduling policies required for the timely execution of those policies.

A control domain generally has both physical and logical connotations, but is application specific. The D/OS provides middleware services relevant to developing, configuring and managing the suite of applications required to manage the customer's plant. These services include:

- configuration services ..... PCS configuration specification and management services
- domain databases.... PCS problem domain configuration services
- operator interfaces ..... PCS man-machine interface environment and tools
- event management.... PCS system-wide event (alarm) management services
- directory services ..... PCS name services
- security services ..... PCS security services
- filing services ..... PCS file system services
- internetworking ..... PCS network interface services
- archiving services ..... PCS activity and data logging services
- reporting services ..... PCS trending and reporting services
- control policies ..... PCS control policy specification services
- transnode scheduling..... PCS resource scheduling services
- fault recovery ..... PCS fault isolation and recovery services
- control events..... PCS event response policies and mechanisms

Many of these services will be applications that reside on the "guest nodes" of the PCS, since they are services that are either required during the development and commissioning of the customer's system, or are not engaged in the direct control of the plant and can run with relaxed availability requirements. Hence, these D/OS services are resident on the PCS host multicomputer, but occupy resources on only a subset of the nodes of the system. The particular nodes engaged in supporting these services, versus the node_set responsible for hosting the actual plant control policies, are determined at configuration time and are dynamically modified under faults, forced reconfigurations, and dynamic "what if" conditions.

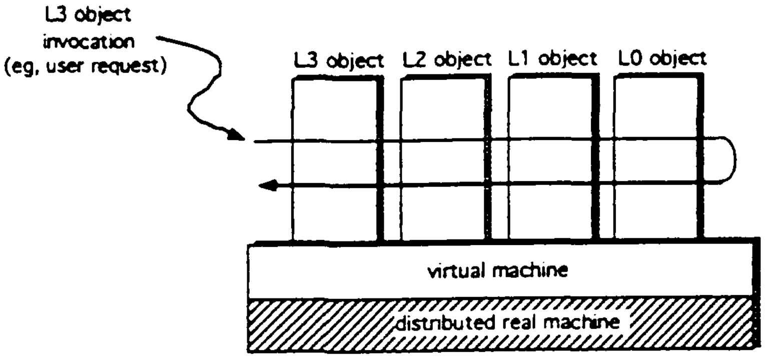
4.3 Control Application Level Semantics

The semantics of the application layer of the PCS are based on i) classical linear, sequential, sampled-data control [Houpis92], ii) advanced non-linear, stochastic, optimal estimation, filtering, and control [Isermann91]; and iii) controls based on heuristics [Slagle71] and such non-traditional mechanisms as neural and fuzzy logic controllers [Kosko92]. These semantics are quite rich in both traditional usage and theory, and this paper is not the forum for a tutorial review.

In the situation where control policies require the services (resources) of applications running outside of the execution domain of the distributed operating system, those services can be mounted on a guest operating system (e.g., NT, VMS or OSF/1) and run as a server on one of the nodes, or one of the processor_sets, of the host.

As an example, a continuous emissions monitoring (CEM) application might engage the services of plantwide, supervisory, regulatory, and transducer objects of the PCS. These objects contain (possibly multi-threaded) tasks that carry work on behalf of a number of concurrently executing applications. The CEM application might begin its life by invoking activity in a L3 object (i.e., the application-level thread) which manifests itself as possibly many parallel Mach execution threads. The CEM application progresses along this two dimensional trajectory in time, sequentially through the CEM application (a la the client-server paradigm), and concurrently along the various control policies as instantiated in the subservient control objects (a la peer-to-peer).

The figure below is a simplistic depiction of the "application thread" meandering through the distributed objects that govern PCS control policies.



This figure conveys the peer-level semantics of control policy objects whose (ideally, provably correct) operations are restricted to a given problem domain. This model is being used to test the concepts of dynamic replacement of running control policies by substituting, in real-time, the control policies governing various processes. This dynamic control policy feature is critical to fault isolation, load balancing under upset conditions, policy "what ifs" based on high fidelity simulations against the running plant, and other related applications requirements.

5. Conclusions

Within the framework presented here, there are a number of open conceptual, implementation, and technology-related issues. They represent ongoing efforts in defining application domain and platform semantics that are required to realize the next generation of plant control systems. The business drivers for integrated vertical applications in an open real-time computing environment clearly dictate capabilities not found in current distributed system platforms, but which we believe are enabled by the computing environments defined in the cited work. Key to these new computing technologies is the body of work related to the real-time distributed resource management architecture as defined in the Mach [Rashid86], Archons [Northcutt87], and Clouds [Dasgupta88] experience of the late 1980's. This work has led to the merging of capabilities in the standards-based OSF environment, especially the OSF kernel work at CHPC, OSF Advanced Development, and elsewhere.

We believe the experiences gained to date justify the development of distributed real-time platforms based on non-traditional scheduling policies. The key feature of a new class of policies is their basis in the distributed threads model, and the ability to associate policy with all the resources of a computation, even if such policies exist across processors in a processor_set. This trans_node scheduling capability is critical to the implementation of adaptable configurations that can be restructured under the stochastic nature of real-time mission critical computations.

6. References

[Anderson87] Anderson, D.P., Ferrari, D., Rangan, P.V., and Tzou, S-Y., "The DASH Project: Issues in the Design of a Very Large Distributed System", Report No. UCB/CSD 87/338, UC Berkeley EECS Department, January 1987.

[Baker89] Baker, T.P. and Shaw, A., "The Cyclic Executive Model and Ada", The Journal of Real-Time Systems, Vol. 1, 1989, pp. 7-25.

[Booch91] Booch, G., Object Oriented Design with Applications, Benjamin/Cummings Publishing Co., 1991.

[Cheng88] Cheng, S-C. and Stankovic, J.A., "Scheduling Algorithms for Hard Real-Time Systems - A Brief Survey", Tutorial on Hard Real-Time Systems, IEEE, 1988.

[Clark92] Clark, R.K., Jensen, E.D., and Reynolds, F.D., "An Architectural Overview of the Alpha Real-Time Distributed Kernel", Proceedings of the USENIX Workshop on Micro-Kernels and other Kernel Architectures, USENIX Association, April 1992, pp. 127-146.

[Dasgupta88] Dasgupta, P., LeBlanc, R.J., and Appelbe, W.F., "The Clouds Distributed Operating System: Functional Description, Implementation Details and Related Work", Proceedings of the 8th International Conference on Distributed Computing Systems, June 1988.

[Ehling67] Ehling, E.H., Range Instrumentation, Prentice-Hall, 1967.

[Hennessy90] Hennessy, J.L. and Patterson, D.A., Computer Architecture: A Quantitative Approach, Morgan Kaufmann Publishers, 1990.

[Houpis92] Houpis, C.H., Lamont, G.B., Digital Control Systems - Theory, Hardware, Software, McGraw-Hill, Inc, 1992.

[Isermann91] Isermann, R., Digital Control Systems, Vol II, Stochastic Control, Multivariable Control, Adaptive Control, Applications, Springer-Verlag, 1991.

[Jensen85] Jensen, E.D., Locke, C.D., and Tokuda, H., "A Time-Driven Scheduling Model for Real-Time Operating Systems", Proceedings of the IEEE Symposium on Real-Time Systems, IEEE, 1985.

[Jensen90] Jensen, E.D. and Northcutt, J.D., "Alpha: A Non-Proprietary OS for Large, Complex, Distributed Real-Time Systems", ACM Press, 1990.

[Jensen92] Jensen, E.D., "A New Perspective on Realtime Computing for the 1990's, Draft Revision June 24, 1992", Digital Equipment Corporation, 1992.

[Jones86] Jones, M.B. and Rashid, R., "Mach and Matchmaker: Kernel and Language Support for Object-Oriented Distributed Systems", OOPSLA '86 Proceedings, September 1986.

[Lehoczky89] Lehoczky, J., Sha, L., and Ding, Y., "The Rate Monotonic Scheduling Algorithm: Exact Characterization and Average Case Behavior", IEEE, 1989.

[Liskov82] Liskov, B., "On Lingusitic Support for Distributed Programs", IEEE Transactions on Software Engineering, May 1982.

[Liu73] Liu, C.L., and Layland, J.W., "Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment", Journal of the Association for Comptuing Machinery, Vol. 20, No. 1, January 1973, pp. 46-61.

[Loepere92] Loepere, K., "Mach 3 Kernel Principles", Open Software Foundation and Camegie Mellon University, OSF Research Institute, July 1992.

[Northcutt87] Northcutt, J.D., Mechanisms for Reliable Distributed Real-Time Operating Systems - The Alphas Kernel, Academic Press, 1987.

[OSF92] Open Software Foundation, Introduction to OSF™ DCE, Prentice Hall, 1992.

[PEI90] Protocol Engines, Inc., The eXpress Transport Protocol Specification, Version 3.5, August 1990.

[Rashid86] Rashid, R., "Threads of a New System", Unix Review, August 1986.

[Rosier88] Rosier, M.,10 et al, "Overview of the Chorous Distributed Operating Systems", Proceedings of the Workshop on Micro-kemels and other Kernel Architectures, The Usenix Association, April 1992, pp. 39-69.

[Sha83] Sha, L., Jensen, E.D., Rashid, R., and Northcutt, J.D., "Distributed Cooperating Processes and Transactions", Proceedings of the ACM Symposium on Data Communication Protocol Architectures, March 1983.

[Sha90a] Sha, L. and Goodenough, J.B., "Real-Time Scheduling Theory and Ada®", IEEE Computer, April 1990.

[Sha90b] Sha, L., Rajkumar, R., and Lehoczky, J.P., "Priority Inheritance Protocols: An Approach to Real-Time Synchronization", IEEE Transactions on Computers, Vol. 39, No. 9, September 1990.

[Schantz86] Schantz, R.E., Thomas, R.H., and Bono, G., "The Architecture of the Cronus Distributed Operating System", Proceedings of the 6th International Conference on Distributed Computing Systems, IEEE, 1986.

[Shipman92] Shipman, S., Teller, M.J., and Paciorek, N., "Mach/RT Kernel Interfaces, Draft Specifications, Revision 1", Center for High Performance Computing, Worcester Polytechnic Institute, 1992.

[Slagle71] Slagle, J.R., Artificial Intelligence: The Heuristic Programming Approach, McGraw-Hill, 1971.

[Smook89] Smook, G.A., Handbook for Pulp & Paper Technologists, TAPPI, 1989.

[Zlotnick91] Zlotnick, F., The POSIX.1 Standard, A Programmer's Guide, Benjamin/Cummings, 1991.

KEY ISSUE

Computational Model

- Should define functionality, concurrency, termination, communication, (refinement).
- Should contain attributes to represent requirements such as release time, period, deadlines on computation and communication, utility/benefit etc.
- Should contain attributes to represent derived properties such as worst case resource usage, completion times (worst case or probabilistic), allocations, priorities etc.
- Should allow decomposition, with the attributes defined at appropriate levels.
- Should recognise that all computations/ communications take time.
- Should facilitate analysis of final (and incomplete) designs.

- Should allow the non-determinacy in the environment to influence the dynamic behaviour of the system.

- Should not force premature commitment to configuration issues.

Above requirements lead to an asynchronous model of interaction.

Two Design Methods have been built upon this computational model:

- HRT-HOOD — a structured method

- TAM — a formal method

HRT-HOOD

Gives explicit support to common object classes:

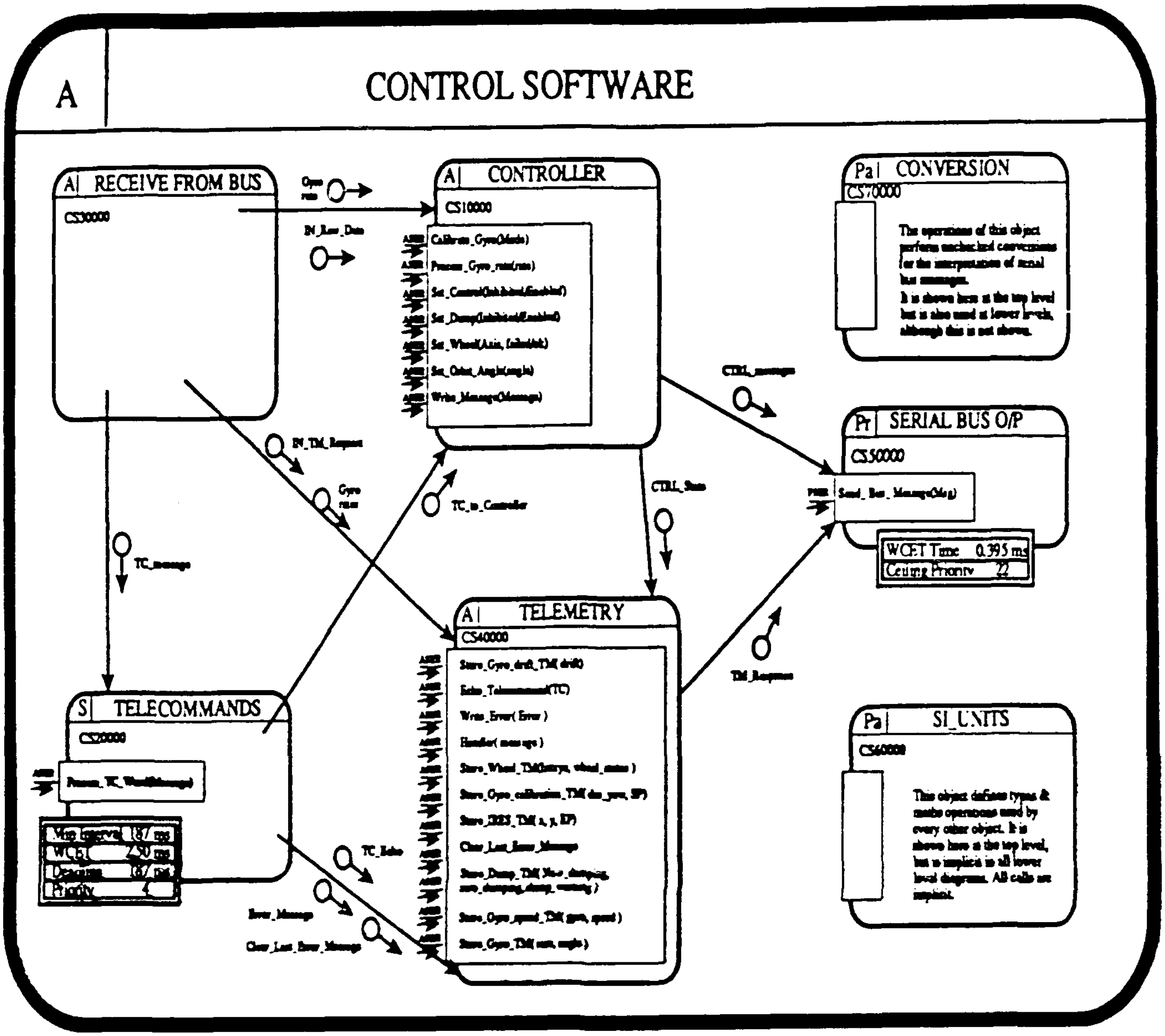
Active
Periodic
Sporadic
Passive
Protected

Periodic and sporadic objects contain a single thread and communicate via passive and protected objects.

Attributes define deadlines, offsets, worst case response times etc.

Rules of decomposition force the crucial objects is a system to be predicatable in their worst case behaviour.

Transformation are available from HRT-HOOD designs to Ada 9X code analysed (off-line) using static allocation (of objects to processors), and static priorities with immediate ceiling priority inheritance


A
CONTROL SOFTWARE
A RECEIVE FROM BUS
CS30000
Gyro rate
IN_Raw_Data
A CONTROLLER
CS10000
ASER Calibrate_Gyro(Mode)
ASER Process_Gyro_rate(rate)
ASER Set_Control(Inhibited/Enabled)
ASER Set_Dump(Inhibited/Enabled)
ASER Set_Wheel(Axis, Inhibit)
ASER Set_Offset_Angle(angle)
ASER Write_Message(Message)
Pa CONVERSION
CS70000
The operations of this object perform unchecked conversions for the interpretation of serial bus messages.
It is shown here at the top level but is also used at lower levels, although this is not shown.
CTRL_message
Pr SERIAL BUS O/P
CS50000
PSER Send_Bus_Message(Msg)
WCET Time 0.395 ms
Ceiling Priority 22
IN_TM_Request
Gyro rate
TC_to_Controller
CTRL_State
TC_message
TM_Response
A TELEMETRY
CS40000
ASER Store_Gyro_drift_TM(drift)
ASER Echo_Telecommand(TC)
ASER Write_Error(Error)
ASER Handler(message)
ASER Store_Wheel_TM(entry, wheel_status)
ASER Store_Gyro_calibration_TM([illegible])
ASER Store_IRES_TM(x, y, EP)
ASER Clear_Last_Error_Message
ASER Store_Dump_TM([illegible])
ASER Store_Gyro_speed_TM(gyro, speed)
ASER Store_Gyro_TM(axis, angle)
S TELECOMMANDS
CS20000
ASER Process_TC_Word(Message)
Min Interval 187 ms
WCET 250 ms
Deadline 187 ms
Priority 4
TC_Echo
Error_Message
Clear_Last_Error_Message
Pa SI_UNITS
CS60000
This object defines types & maths operations used by every other object. It is shown here at the top level, but is implicit in all lower level diagrams. All calls are implicit.

CE_DISK_92-044:CS00000

Last Modified: 08/Aug/92

Figure 3: The Control Software

CONTROLLER

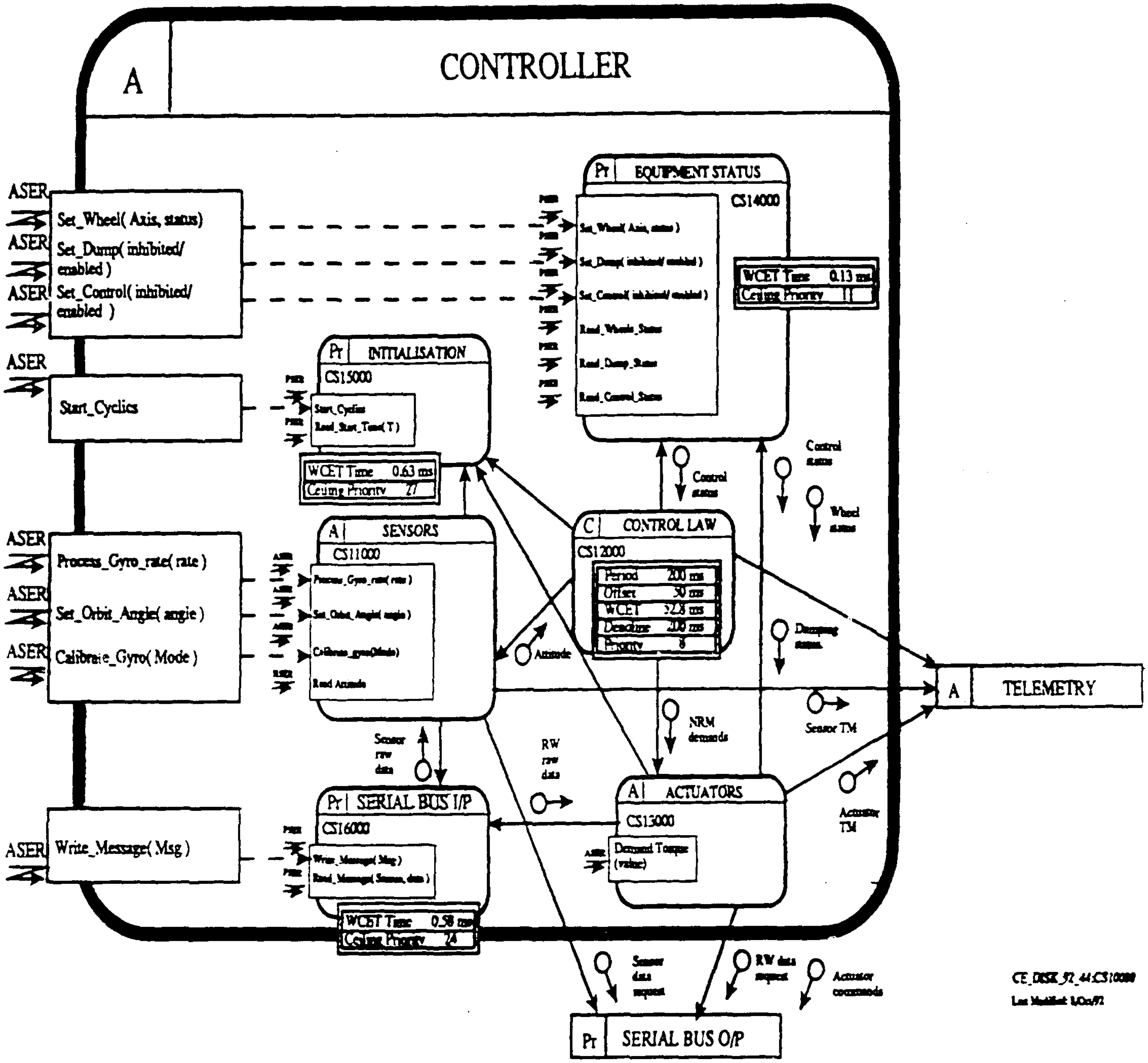


Figure 4: The CONTROLLER Object

Further decomposition of the SENSOR object is shown in Figures 5 to 9.

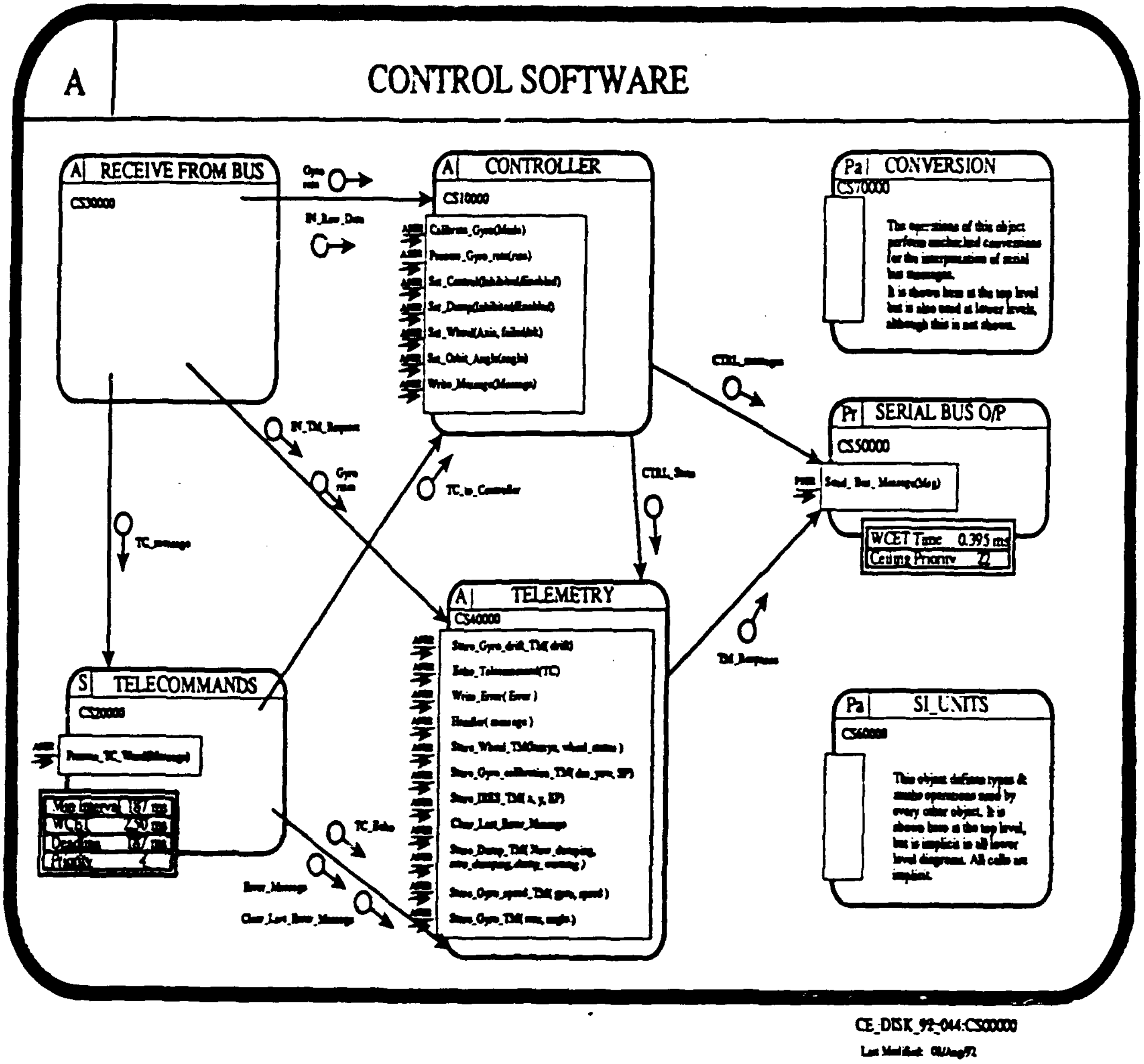


Figure 3: The Control Software

CONTROLLER

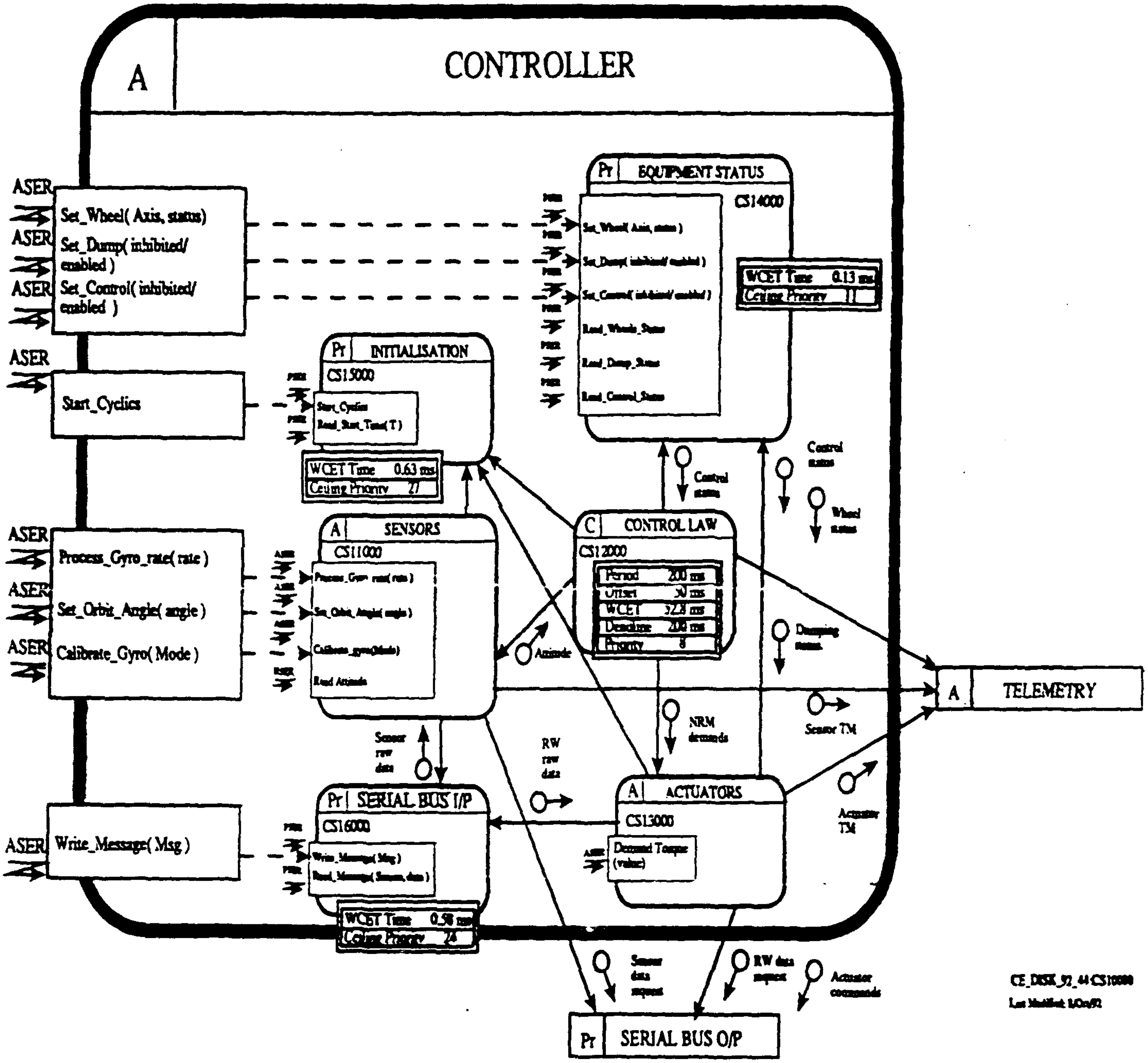


Figure 4: The CONTROLLER Object

Further decomposition of the SENSOR object is shown in Figures 5 to 9.

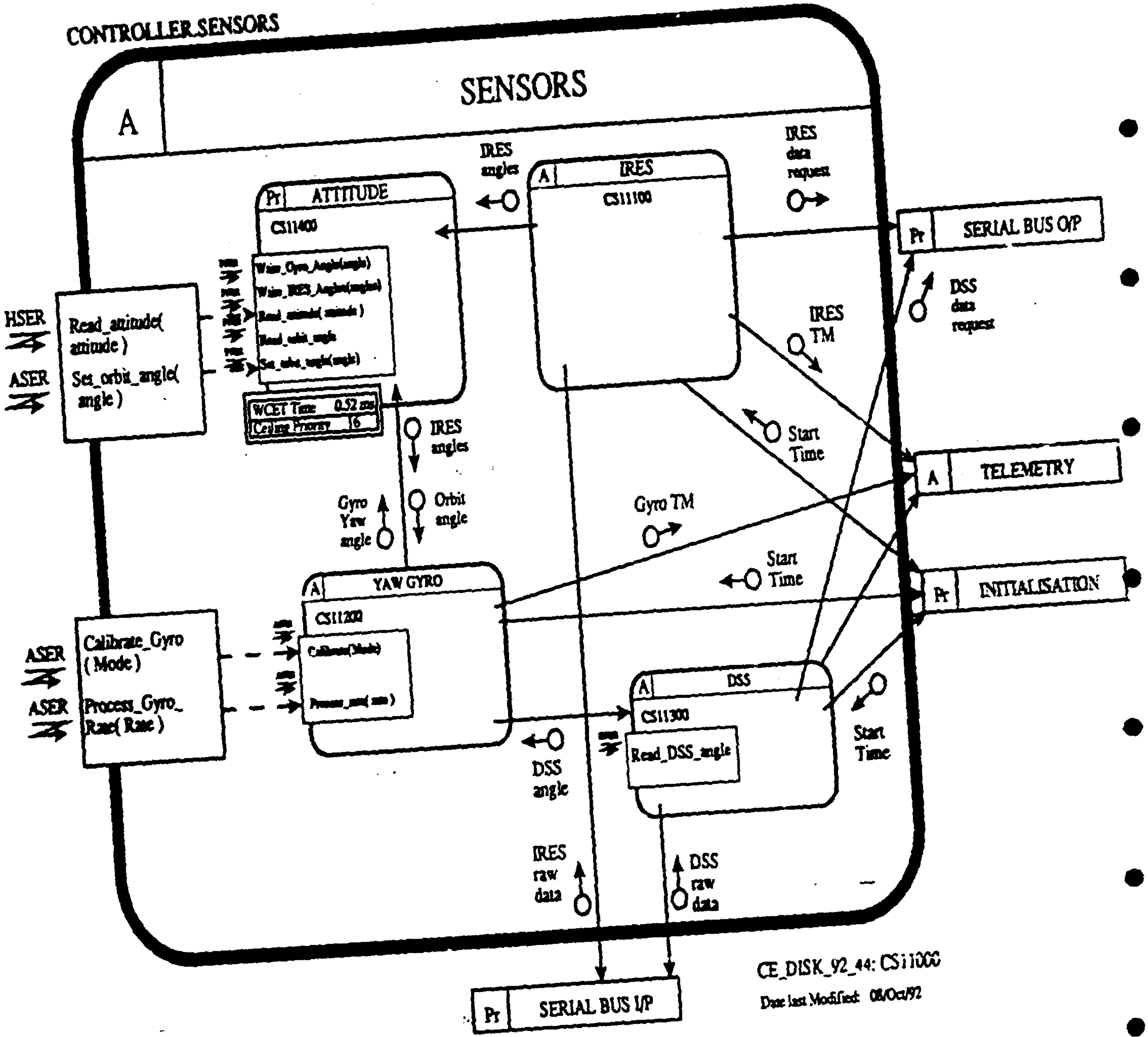


Figure 5: The SENSOR Object

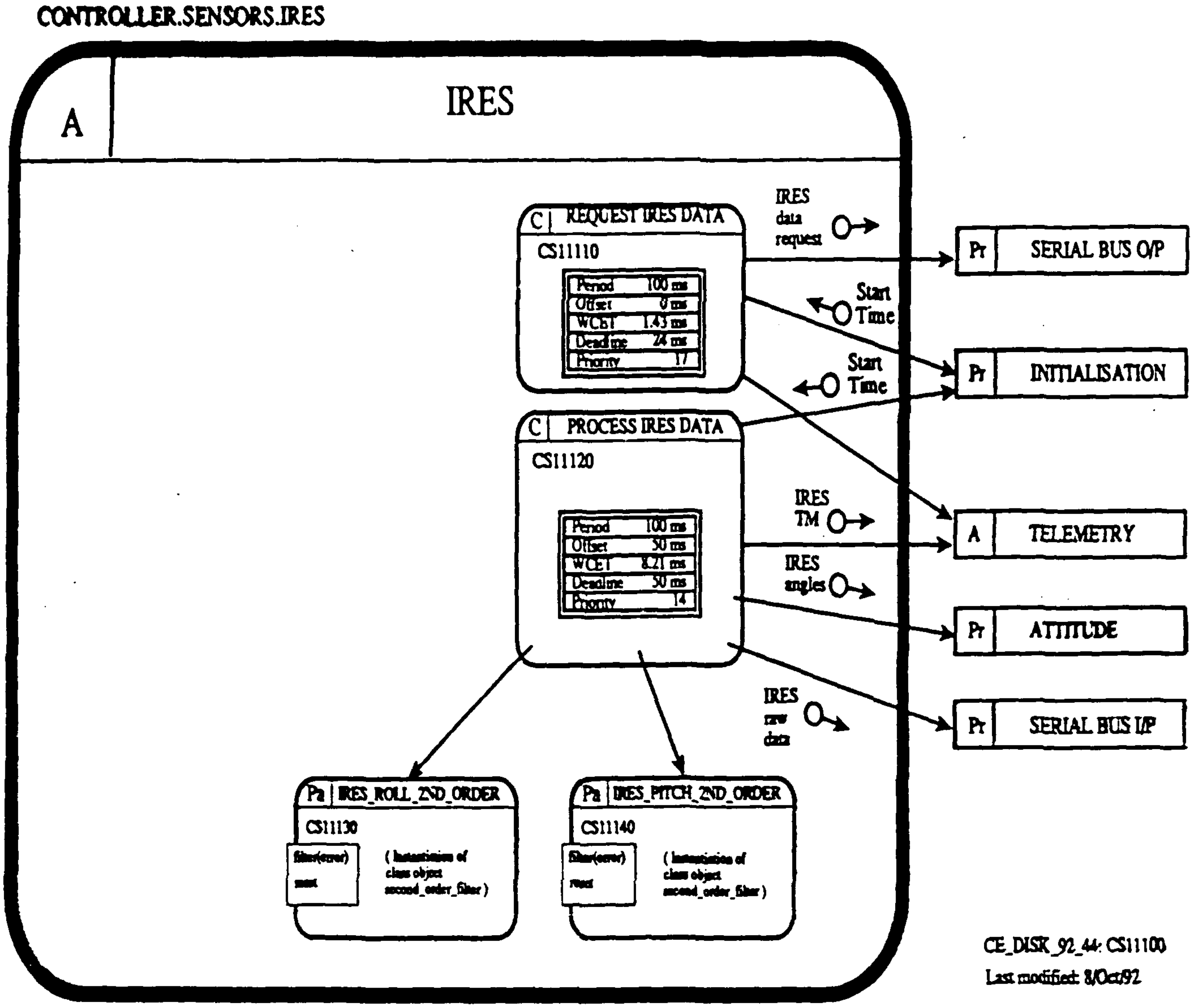
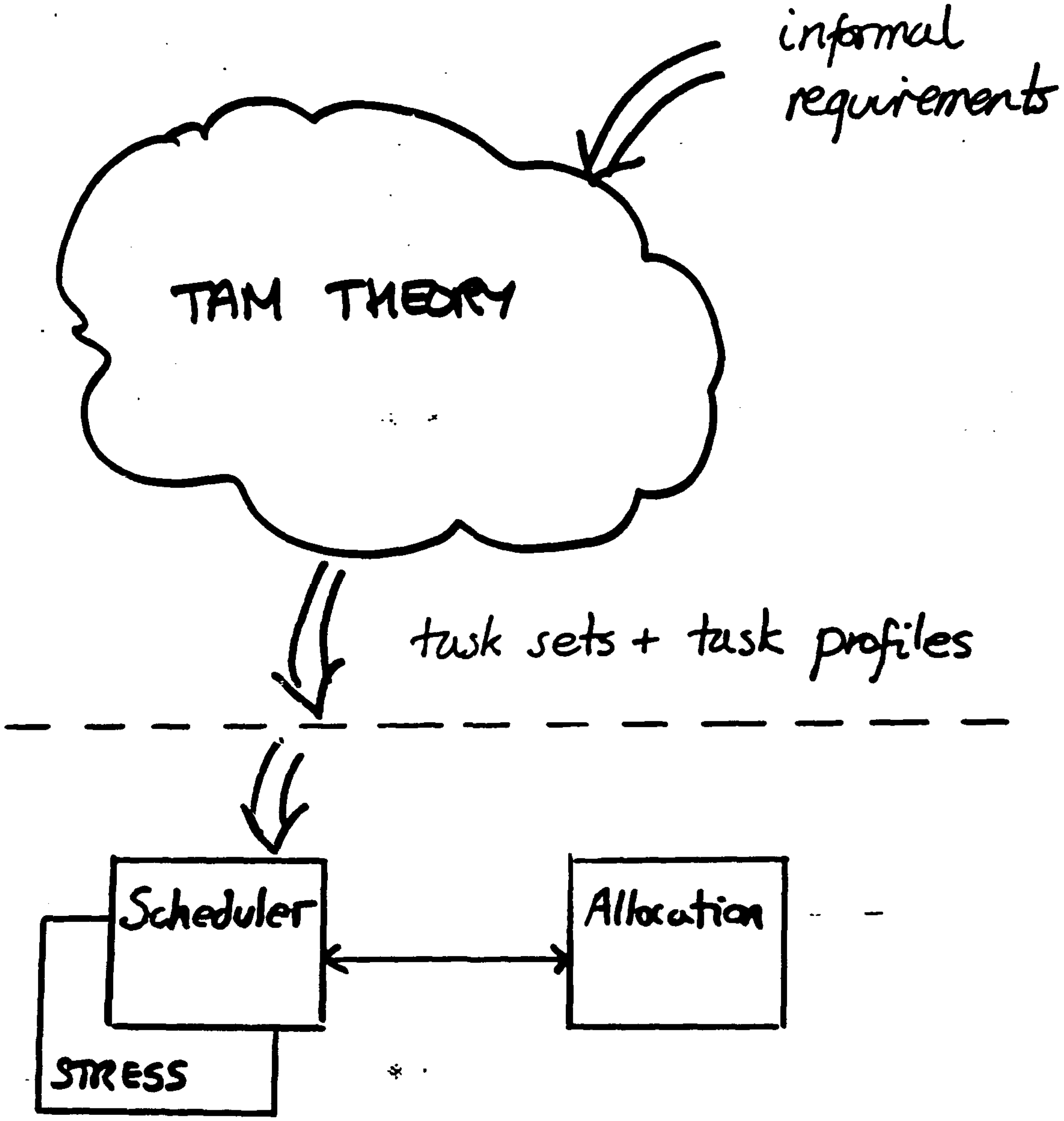


Figure 6: The IRES Object

The IRES sensor controller consists of two precedent constrained cyclic objects with a time offset between the two implementing the required synchronisation. The first object, REQUEST IRES DATA, sends a request to the IRES (via the serial bus). The second object will receive and interpret the sensor values. The relative time offset between the task releases and the deadline of the first object ensures that the sensor device has a chance to respond (at least 30 ms).

TAM and the RTEE



Computational Model (Informal)

Agents

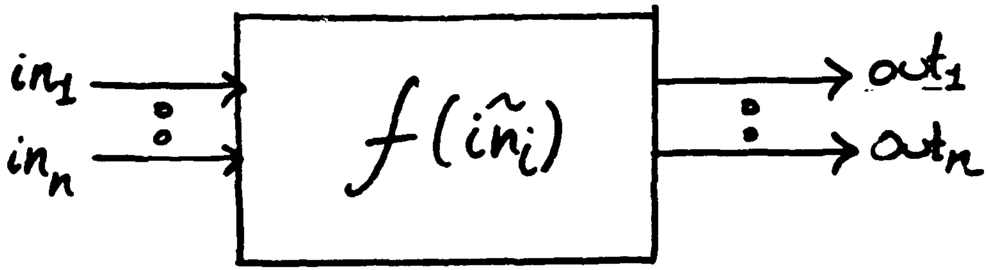
local state variables

input/output shunts

all finite

all timed

Concurrent



Shunts

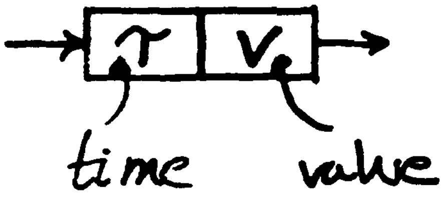
Communication mechanism

asynchronous

single writer

single reader

time stamped on write



Time

Positive integers + ω

TAM Syntax

$x := e$	assignment
$x \Rightarrow s$	output
$x \leftarrow s$	input
$\omega : \Phi$	specification
$(x{:}T)A$	local variable
$A \backslash (s{:}T)$	private shunt
$A \mid B$	concurrent
$\bigsqcup_{i \in I}^{t} g_i \Rightarrow A_i$	guarded
$[n]A$	deadline
$A \rhd_{n}^{s} B$	timeout
$\mu_{n}^{p} A$	iteration
$A \; ; \; B$	sequence

Program? – Where?

$x \leftarrow s \quad \rightsquigarrow \quad$ read(x, s)

$x \rightarrow s \quad \rightsquigarrow \quad$ write(x, s)

$[s]A \quad \rightsquigarrow$

```
deadline S
    do A
    od
```

$\bigsqcup_{\sigma}{}_{i \in I}\, g_i \Rightarrow A_i \quad \rightsquigarrow$

```
within σ
    if g1 then A1 else
    if g2 then A2 else
      ⋮
    if none then skip
nihtiw
```

$(x:T)A \quad \rightsquigarrow$

```
local x of T
in
    A
ni
```

$\mu_n^s A \rightsquigarrow$

```
for i = 1 to n
  do
    [s]A
  od
```

$\mathbb{1} \rightsquigarrow$ skip

$\perp \rightsquigarrow$ abort

$A|B \rightsquigarrow$

```
par
  A
  B
```

$A;B \rightsquigarrow$

```
seq
  A
  B
```

```
$(r_1, r_2 : [\mathrm{Time} \times T])$
    $[1](r_1 \leftarrow in_1 \mid r_2 \leftarrow in_2)$;
    $[9](r_1 = r_2 \Rightarrow$ 'true' $\rightarrow$ out
          $\sqcup_2$
          $r_1 \neq r_2 \Rightarrow$ 'false' $\rightarrow$ out )
```

$\rightsquigarrow$

```
local $r_1$ of $[\mathrm{Time} \times T]$
 in
   local $r_2$ of $[\mathrm{Time} \times T]$
   in
     seq
       deadline 1
         do
           par
               read($r_1$, $in_1$)
               read($r_2$, $in_2$)
         od
       deadline 9
         ~~do~~ within 2
           if $r_1 = r_2$ then write('true', out) else
           if $r_1 \neq r_2$ then write('false', out)
```

Specifications

$$\begin{tikzcd} & \omega : \Phi \arrow[dl] \arrow[dr] & \\ \text{typed frame} & & \Phi_{TAM}^{(t,t')} \text{ formula} \end{tikzcd}$$

All Language constructs are given a <u>semantics</u> by associating formulae

$$[\![\omega : \Phi]\!] \triangleq \underbrace{\text{stable}(\tilde{\omega}, t, t')}_{\downarrow \text{ def}} \wedge \Phi$$

$$\bigwedge_{x \in \tilde{\omega}} \left(\bigwedge_{\sigma \in [t,t']} x@\sigma = x@t \right)$$

$\tilde{\omega}$ is the frame of variables in the scope of $\omega : \Phi$ but not in ω.

Example

$\{table : \mathrm{Seq\ of\ int}\}$:

$$\mathrm{range}(table@t') = \mathrm{range}(table@t)$$

$$\wedge \quad \forall \sigma \in (1..\mathrm{length}(table@t')] \; (\, table@t'(\sigma) \geq table@t'(\sigma - 1))$$

$$\wedge \quad t' - t \leq 4 \cdot \mathrm{length}(table@t)$$

Sort a table of unique entries into decending order in time proportional to length"

Expressive Power?

Mok's RTL.

events are tri-valued variables

$$@(e\uparrow, n) = t \triangleq eet = true$$
$$\wedge \left| \{ m \in [\emptyset, t] : eem = true \} \right| = n$$

$$e(e\downarrow, n) = t \triangleq eet = \neg true$$
$$\wedge \left| \{ m \in [\emptyset, t] : eem = false \} \right| = n$$

Note: this is only one way in which to encode RTL

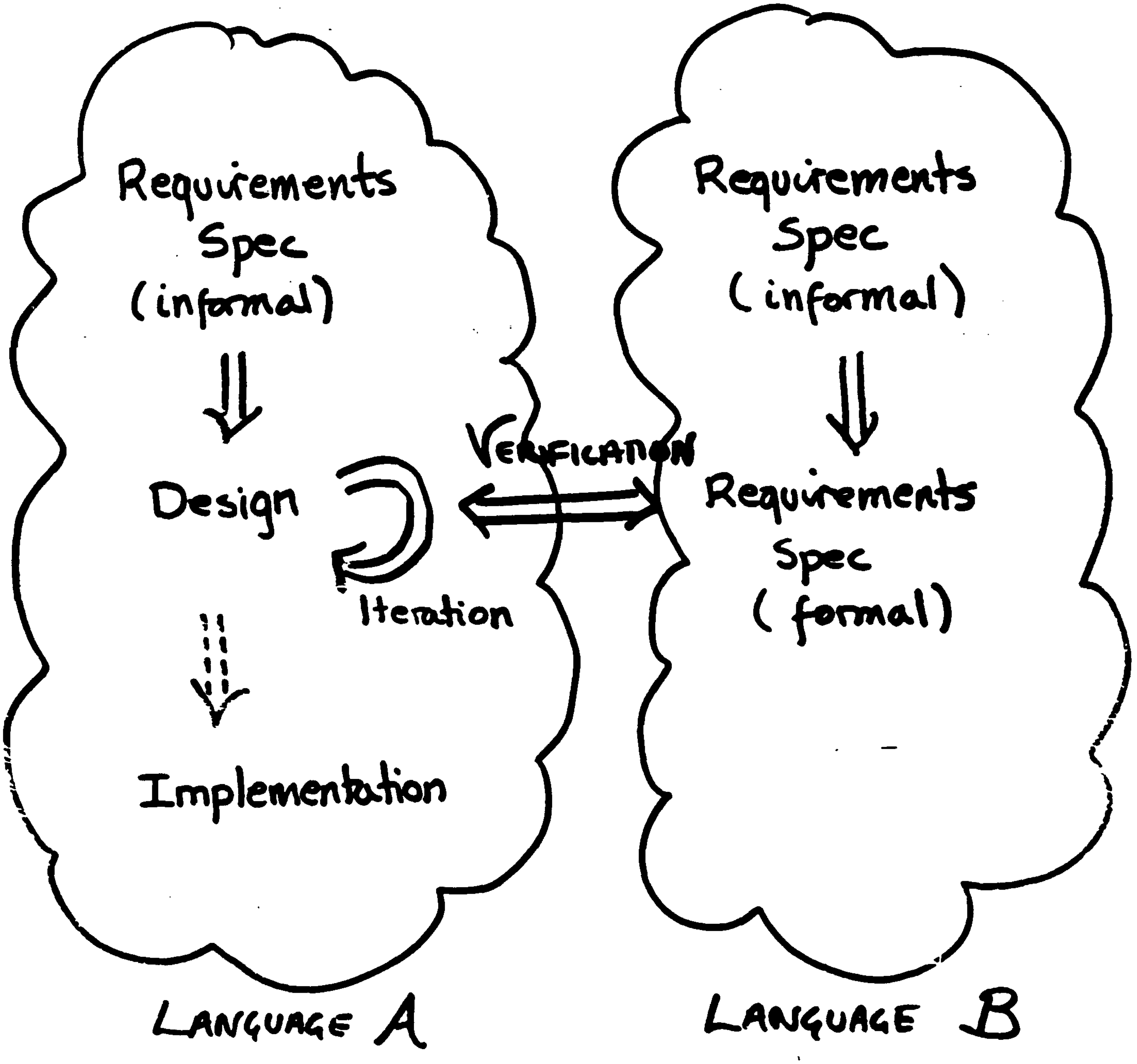
Temporal Logic

Given a predicate $\underline{P}$ on variables $\underline{x}, \underline{y} \cdots$
then

$$\Diamond P(x,y) \triangleq \exists t \in Time\left(P\left[{}^{x@t}/_{x}, {}^{y@t}/_{y}\right]\right)$$

$$\Box P(x,y) \triangleq \forall t \in Time\left(P\left[{}^{x@t}/_{x}, {}^{y@t}/_{y}\right]\right)$$

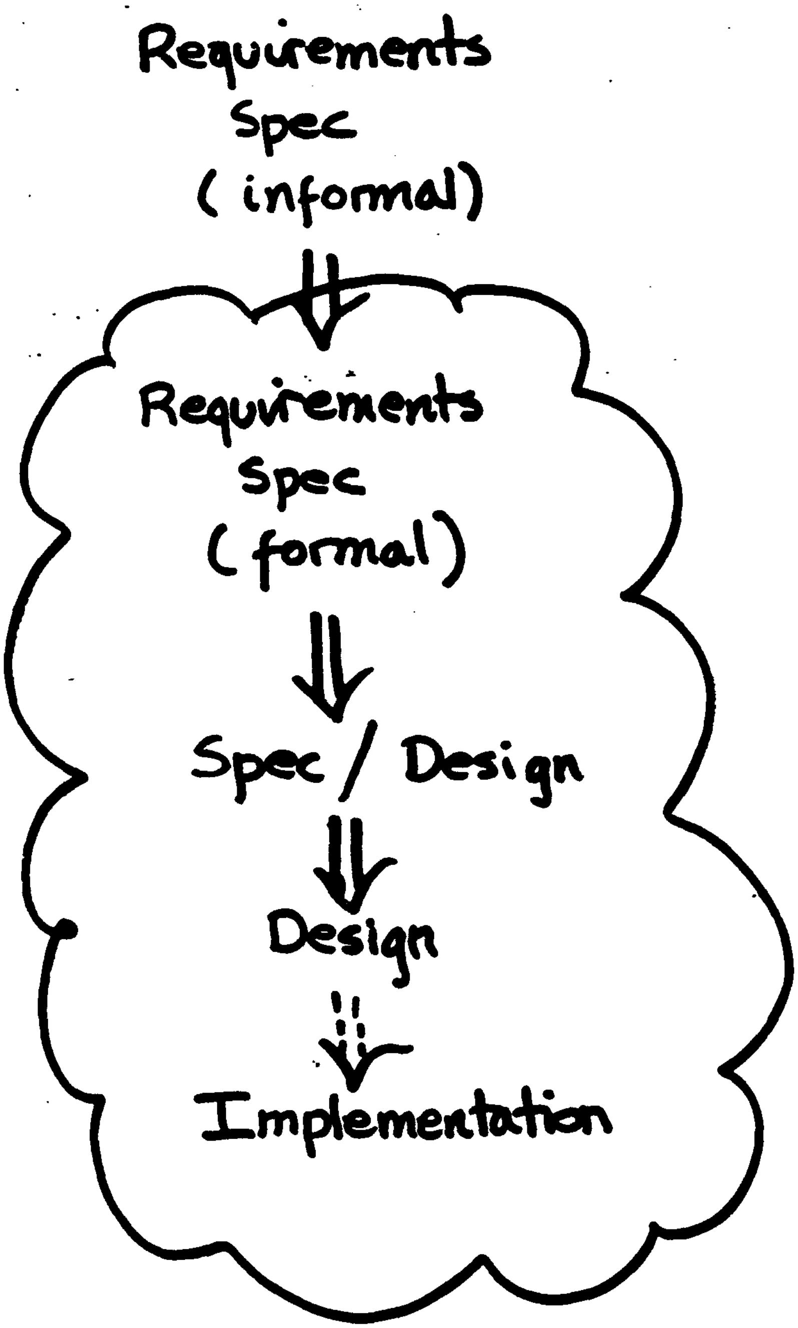
The Complement Method



Complement method : problems

- where does the design come from?
- what guides re-design through failed verification
- what guides verification ? (decidability)
- multiple languages
- two informal translation stages

The Refinement Method



LANGUAGE A

Refinement

Because **all** programs are just formulae we have:

> **Definition**
>
> $$A \sqsubseteq B \quad \text{iff} \quad B \Rightarrow A$$

This gives us a formal fondation for proving rules correct.

Current Status

- First version of HRT-HOOD defined.
- Based on HOOD V3
- Mappings defined for Ada 9X (December 1991)
- Mappings defined for Ada 83 plus ARTEWG CIFO-like entries for added functionality
- UK DRA are supporting a project which will generate CASE tool-support (prototype tools only)
- New version of HRT HOOD this year to address HOOD 3.1 and final version of Ada 9X.

Future NASA Projects

Overview of Projects and Concepts Under Consideration for Future NASA Projects

PRESENTATION TO

Real Time Workshop
Institute for Defense Analyses
Alexandria, Va.
MARCH 15, 1993

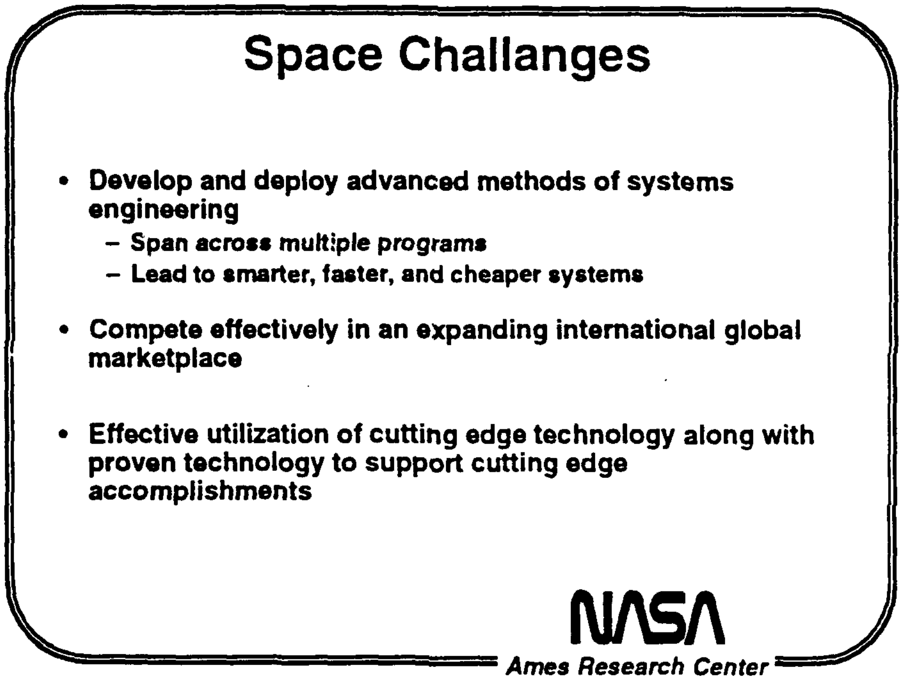
Ed Chevers
Deputy Chief/Information Sciences Division
NASA Ames Research Center

NASA
Ames Research Center

NASA Strategic Intent

- Explore Space
- Support Improvement in Competitiveness
- Reduce Operational Costs

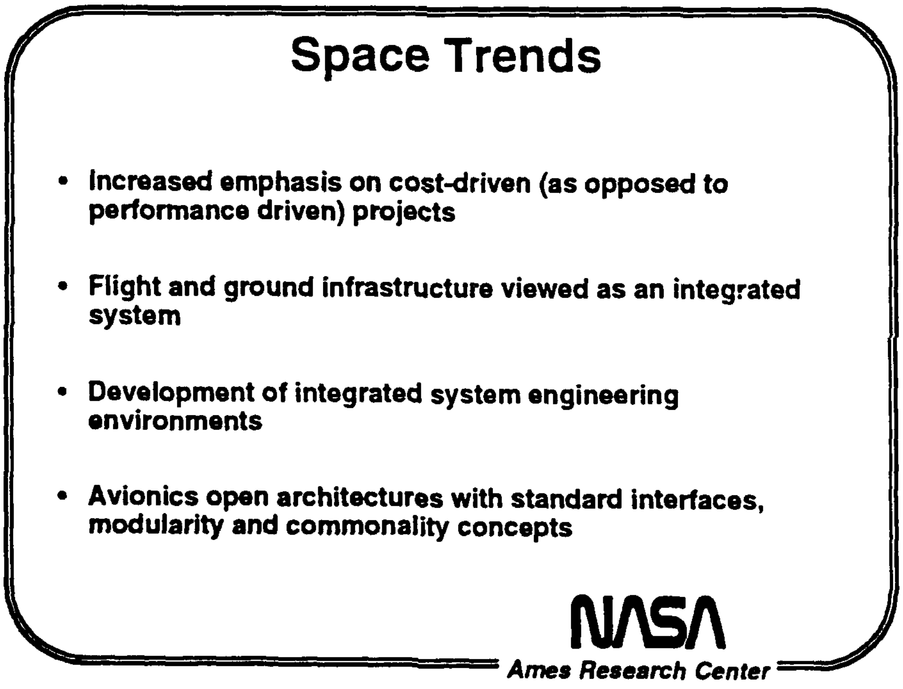
NASA
Ames Research Center



Space Challanges

- Develop and deploy advanced methods of systems engineering
  - Span across multiple programs
  - Lead to smarter, faster, and cheaper systems
- Compete effectively in an expanding international global marketplace
- Effective utilization of cutting edge technology along with proven technology to support cutting edge accomplishments

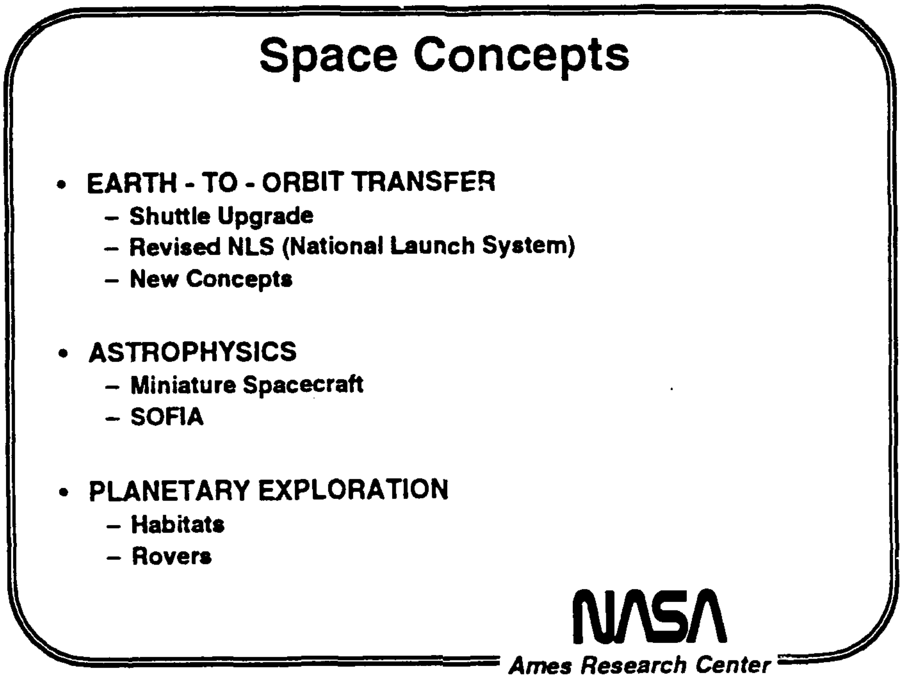
NASA
Ames Research Center



Space Trends

- Increased emphasis on cost-driven (as opposed to performance driven) projects
- Flight and ground infrastructure viewed as an integrated system
- Development of integrated system engineering environments
- Avionics open architectures with standard interfaces, modularity and commonality concepts

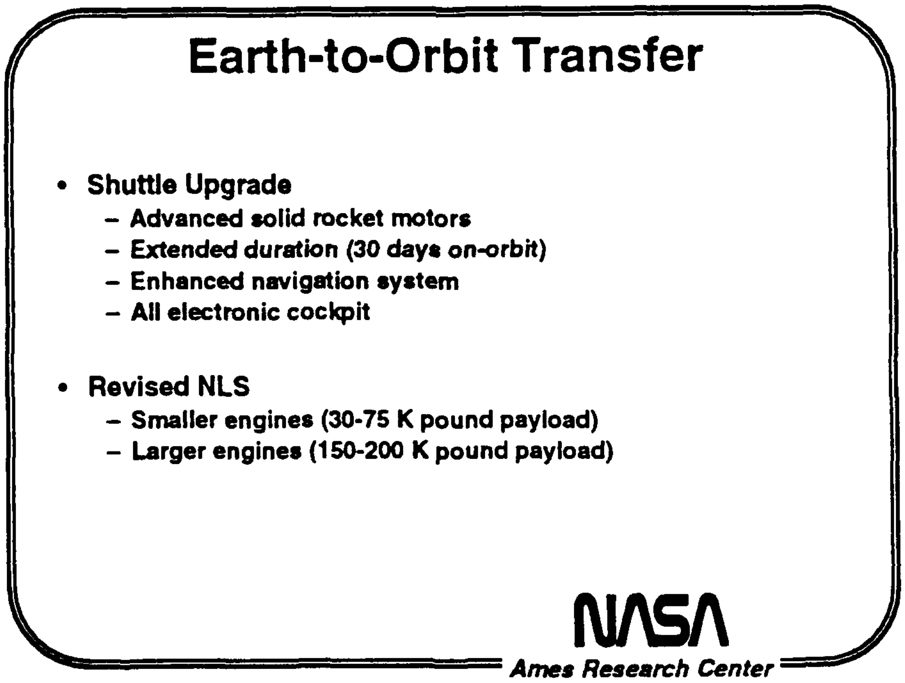
NASA
Ames Research Center



Space Concepts

- EARTH - TO - ORBIT TRANSFER
  - Shuttle Upgrade
  - Revised NLS (National Launch System)
  - New Concepts
- ASTROPHYSICS
  - Miniature Spacecraft
  - SOFIA
- PLANETARY EXPLORATION
  - Habitats
  - Rovers

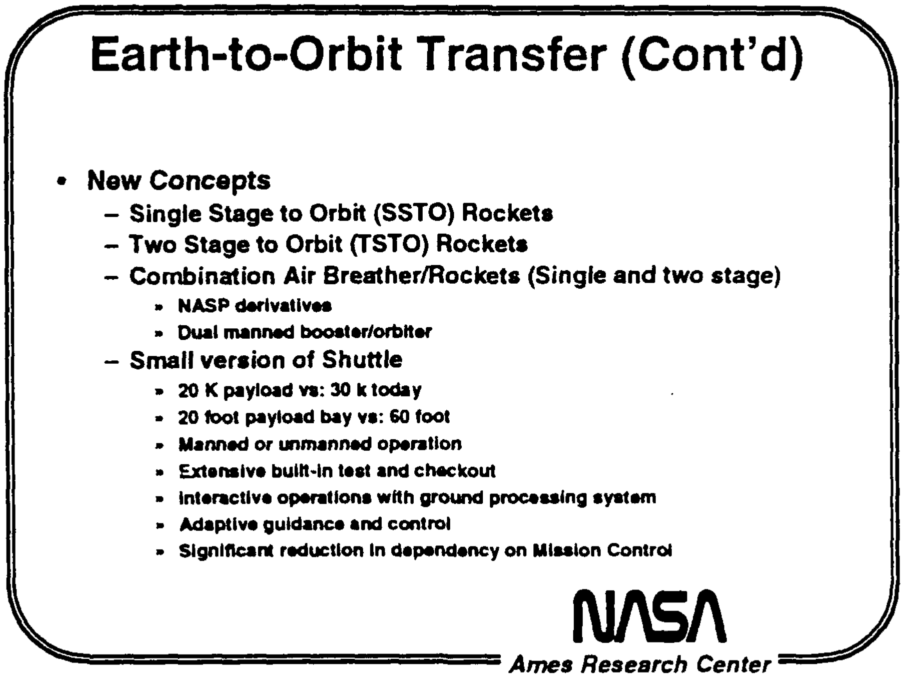
NASA
Ames Research Center



Earth-to-Orbit Transfer

- Shuttle Upgrade
  - Advanced solid rocket motors
  - Extended duration (30 days on-orbit)
  - Enhanced navigation system
  - All electronic cockpit
- Revised NLS
  - Smaller engines (30-75 K pound payload)
  - Larger engines (150-200 K pound payload)

NASA
Ames Research Center

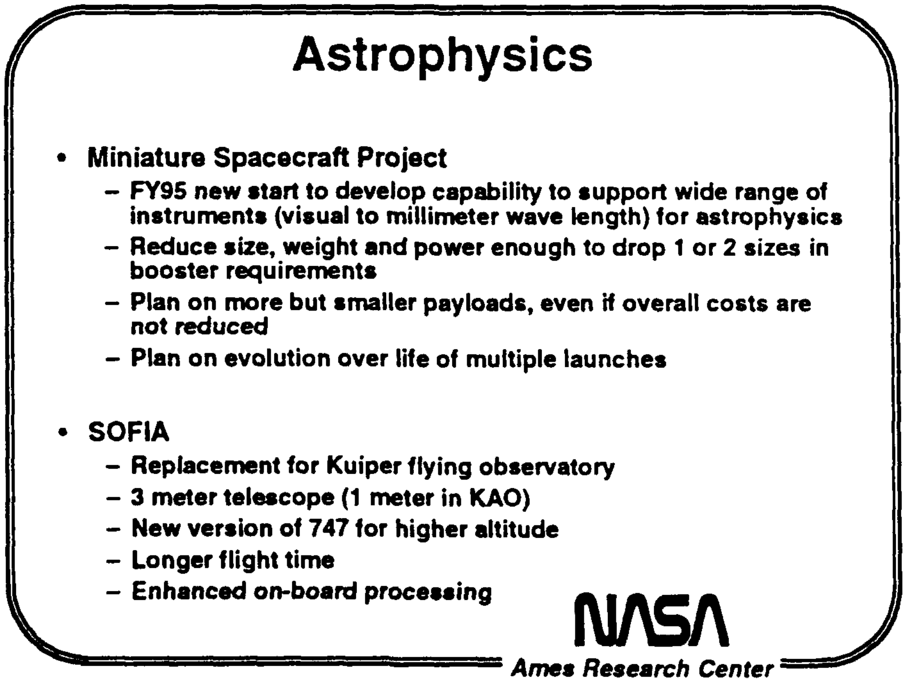


Earth-to-Orbit Transfer (Cont'd)

- New Concepts
  - Single Stage to Orbit (SSTO) Rockets
  - Two Stage to Orbit (TSTO) Rockets
  - Combination Air Breather/Rockets (Single and two stage)
    - NASP derivatives
    - Dual manned booster/orbiter
  - Small version of Shuttle
    - 20 K payload vs: 30 k today
    - 20 foot payload bay vs: 60 foot
    - Manned or unmanned operation
    - Extensive built-in test and checkout
    - Interactive operations with ground processing system
    - Adaptive guidance and control
    - Significant reduction in dependency on Mission Control

NASA

Ames Research Center



Astrophysics

- Miniature Spacecraft Project
  - FY95 new start to develop capability to support wide range of instruments (visual to millimeter wave length) for astrophysics
  - Reduce size, weight and power enough to drop 1 or 2 sizes in booster requirements
  - Plan on more but smaller payloads, even if overall costs are not reduced
  - Plan on evolution over life of multiple launches
- SOFIA
  - Replacement for Kuiper flying observatory
  - 3 meter telescope (1 meter in KAO)
  - New version of 747 for higher altitude
  - Longer flight time
  - Enhanced on-board processing

NASA

Ames Research Center

Planetary Exploration

- Rovers
  - Wheeled vehicles for Moon/Mars
  - Submersible vehicles for Earth's oceans
- Habitats
  - Return to the Moon is stepping stone to real goal: Mars
  - Lunar/Mars habitat should be designed for real time distributed operation from start
  - Ability for in situ sophisticated science data analysis
  - Real time Earth-Planet data analysis/interaction
  - Fully autonomous automated habitat living environment to allow maximum science return
- Ames Human Exploration Demonstration Project (HEDP)

NASA

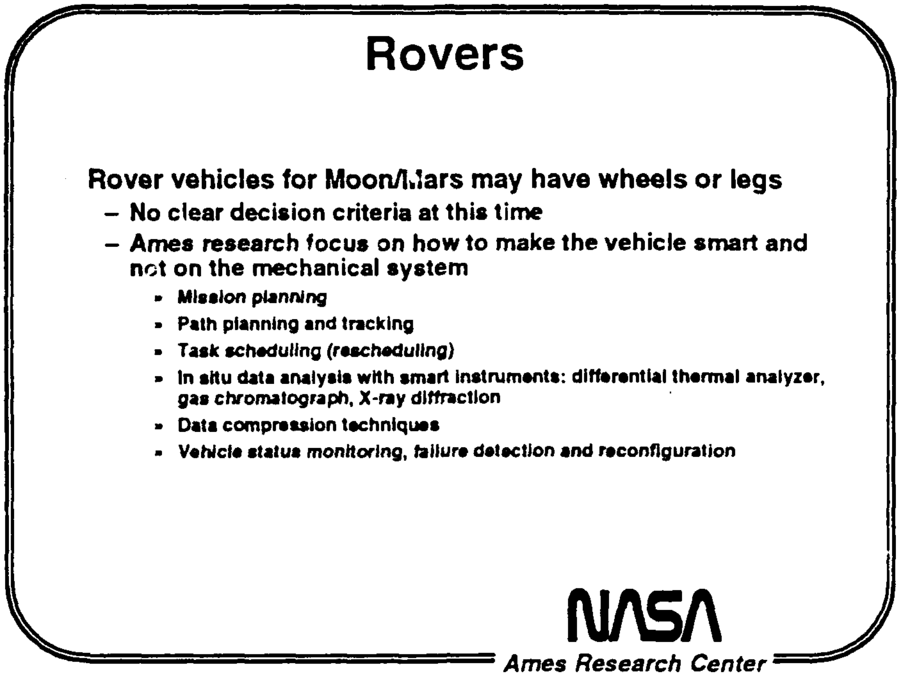
Ames Research Center

Submersibles

- Submersible vehicles for Earth's lakes and bays
  - Relatively complex vehicles being developed for exploration of dry lake beds in Antarctica
    - Helmet mounted display systems for camera control (vehicle next year)
    - Smart instruments for underwater analytical support
  - Similar technology being applied to underwater research in joint Ames/Monterey Bay Aquarium project
    - Multiple sensor fusion
    - 3-D graphics displays
    - Smart manipulators
  - All of these systems are pushing state-of-the-art in real time control and data manipulation
    - Vision processing is primary driver
    - Interaction between multiple scientist and live scene and a virtual scene generated in real time

NASA

Ames Research Center



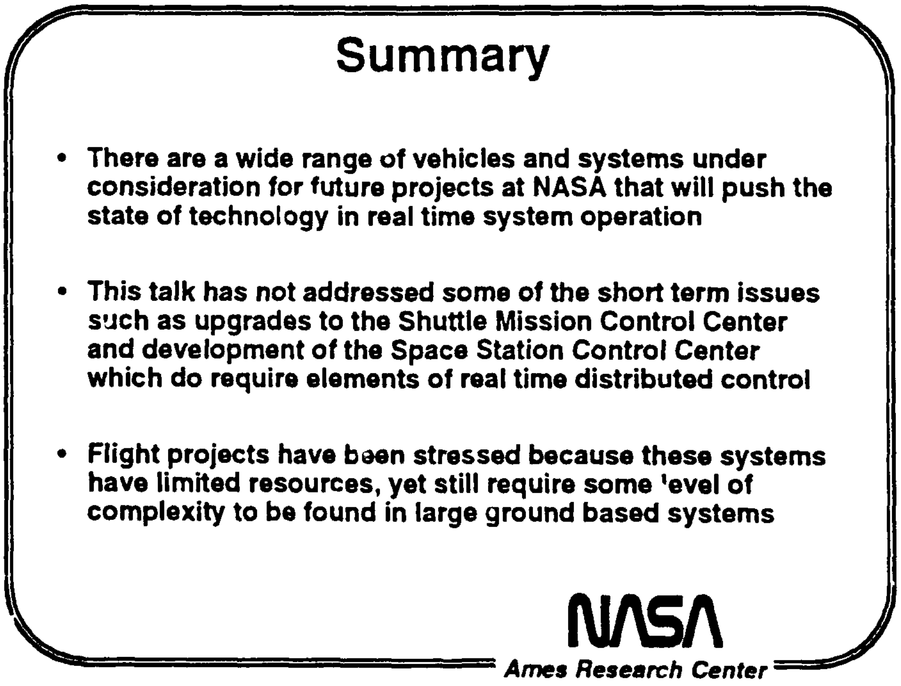
Rovers

Rover vehicles for Moon/Mars may have wheels or legs

- No clear decision criteria at this time
- Ames research focus on how to make the vehicle smart and not on the mechanical system
  - Mission planning
  - Path planning and tracking
  - Task scheduling (rescheduling)
  - In situ data analysis with smart instruments: differential thermal analyzer, gas chromatograph, X-ray diffraction
  - Data compression techniques
  - Vehicle status monitoring, failure detection and reconfiguration

NASA

Ames Research Center



Summary

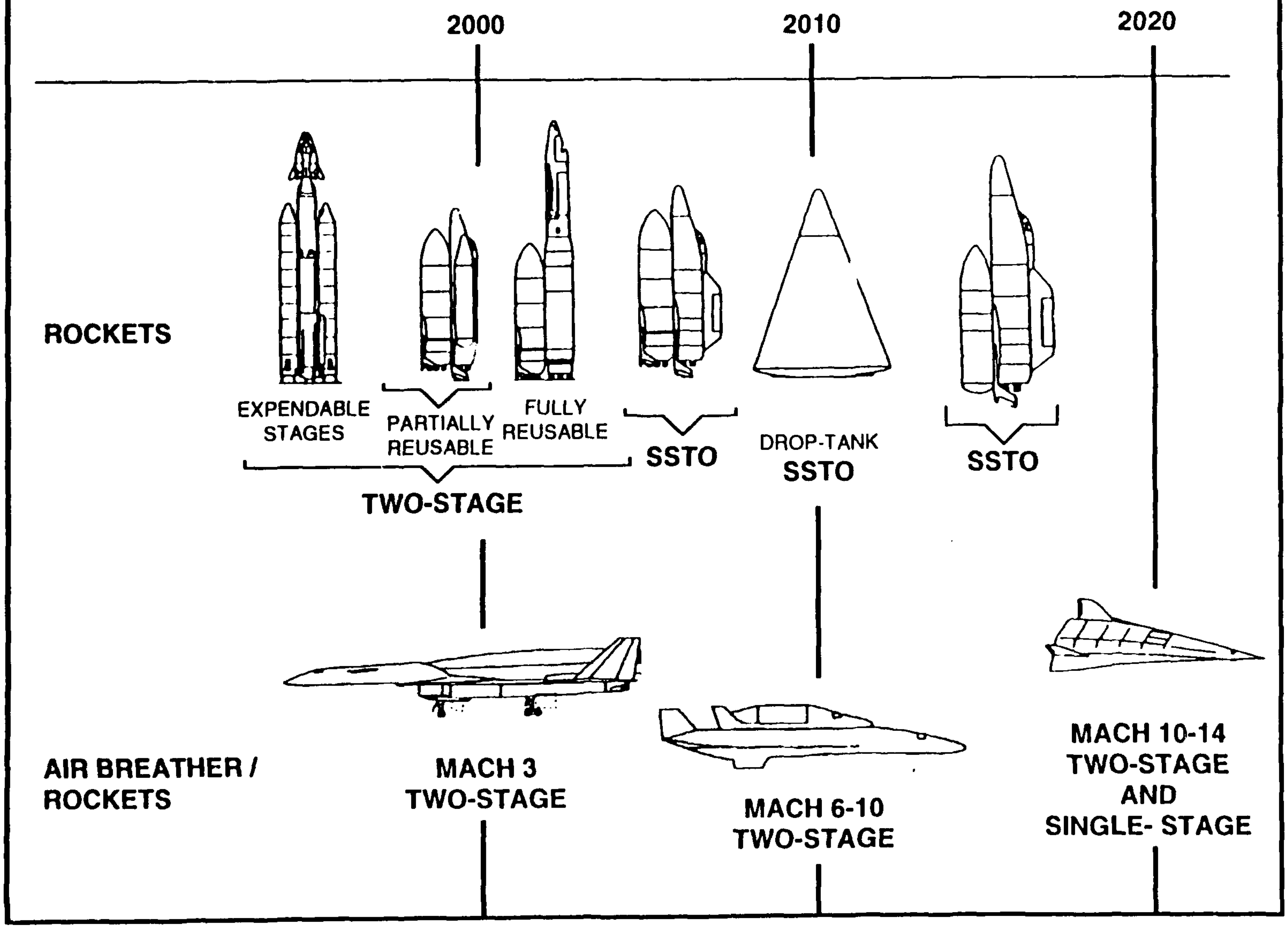
- There are a wide range of vehicles and systems under consideration for future projects at NASA that will push the state of technology in real time system operation
- This talk has not addressed some of the short term issues such as upgrades to the Shuttle Mission Control Center and development of the Space Station Control Center which do require elements of real time distributed control
- Flight projects have been stressed because these systems have limited resources, yet still require some level of complexity to be found in large ground based systems

NASA

Ames Research Center

Earth-to-Orbit Transfer Vehicle Concepts

MEDIUM PAYLOAD (PERSONNEL & CARGO) SPACE VEHICLE CONCEPT OPTIONS



2000

2010

2020

ROCKETS

EXPENDABLE STAGES

PARTIALLY REUSABLE

FULLY REUSABLE

TWO-STAGE

SSTO

DROP-TANK SSTO

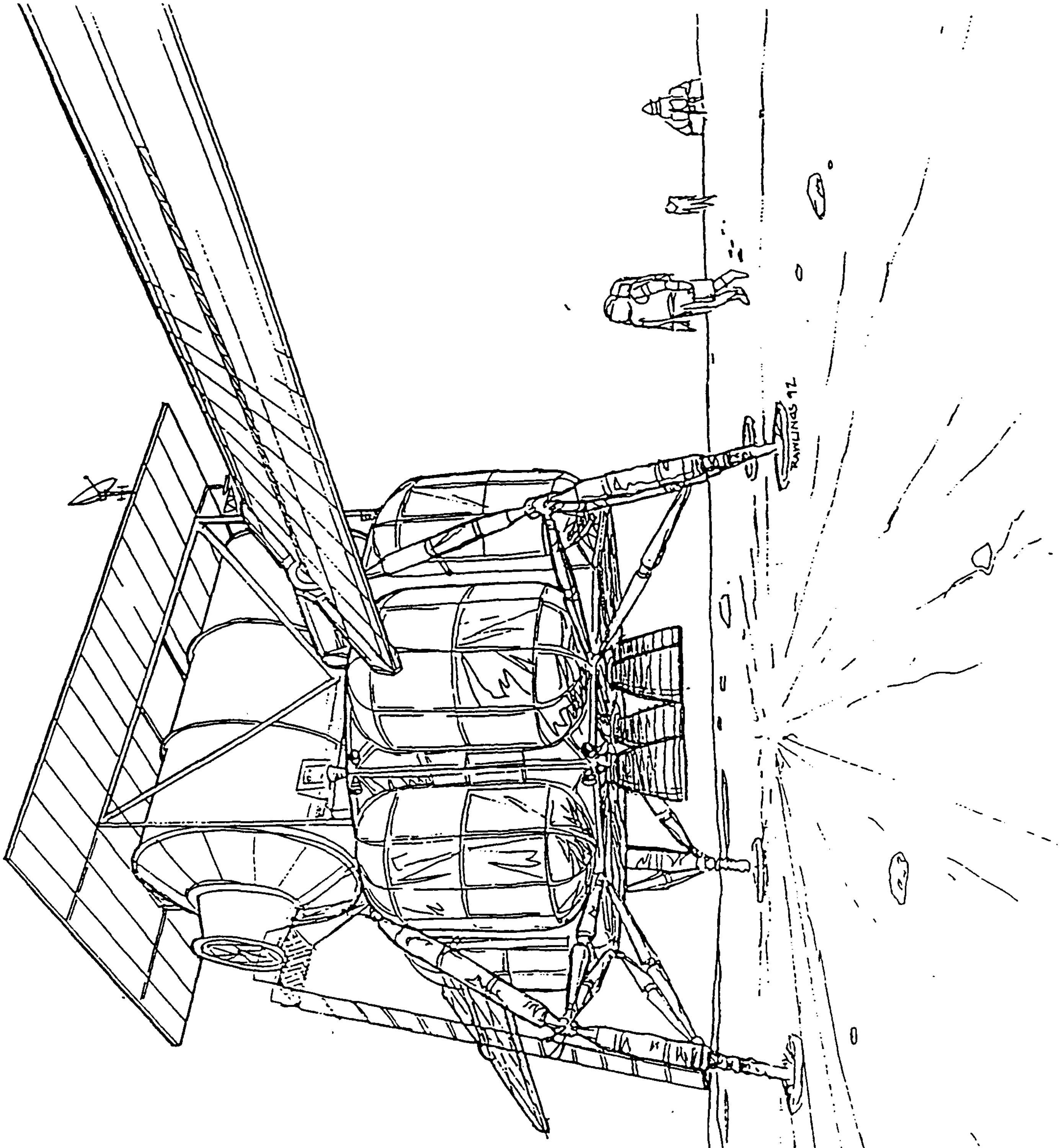
SSTO

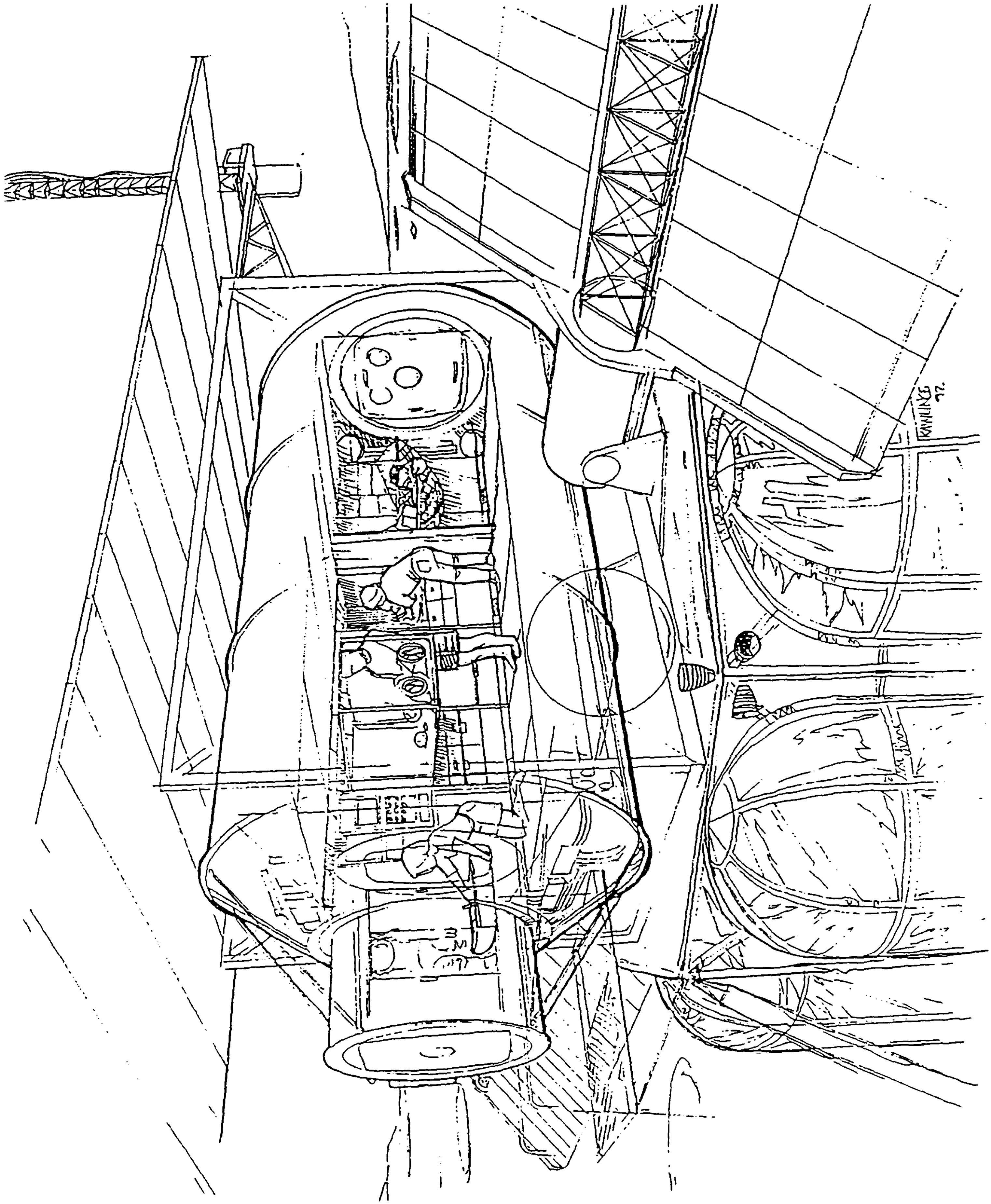
AIR BREATHER / ROCKETS

MACH 3 TWO-STAGE

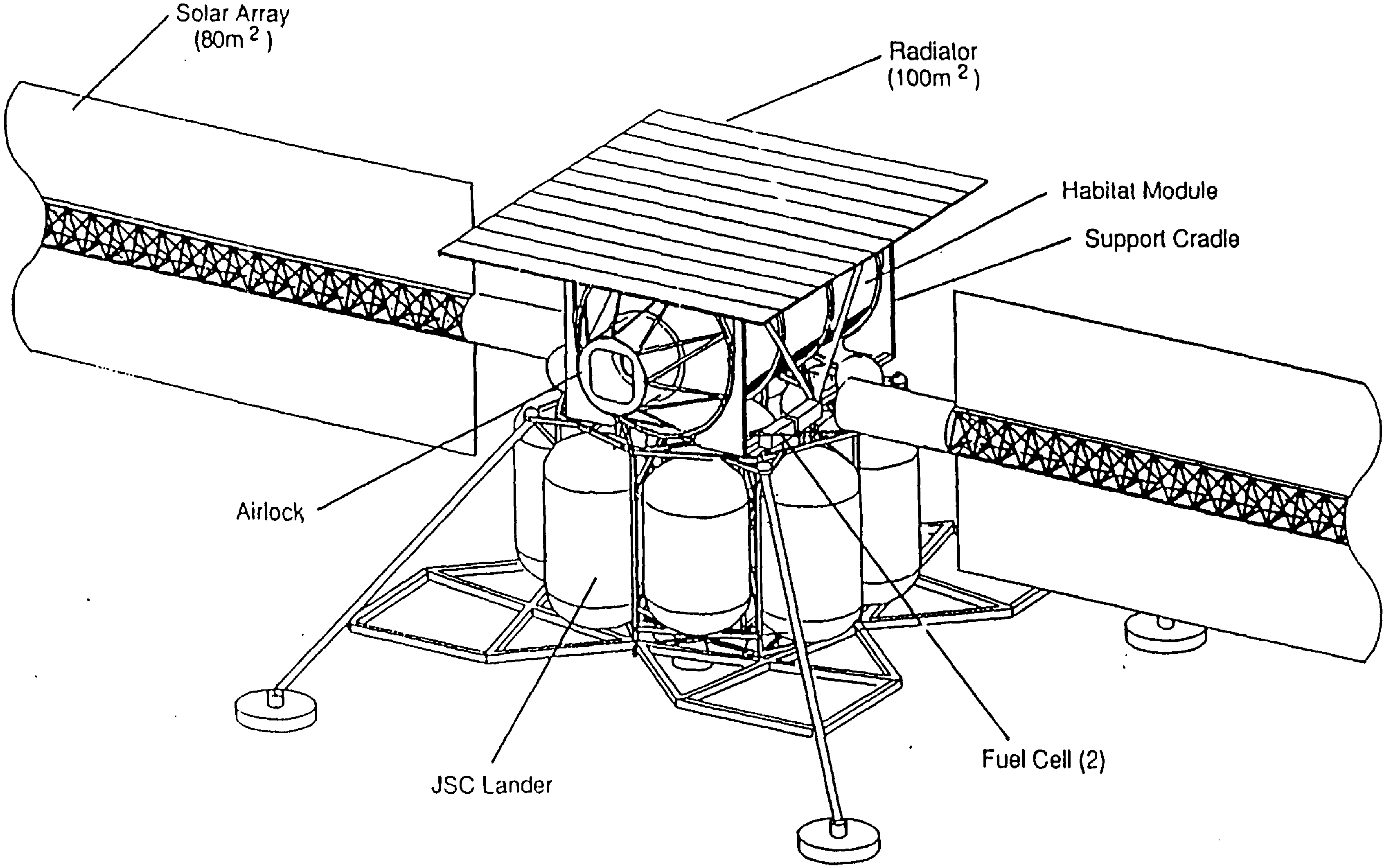
MACH 6-10 TWO-STAGE

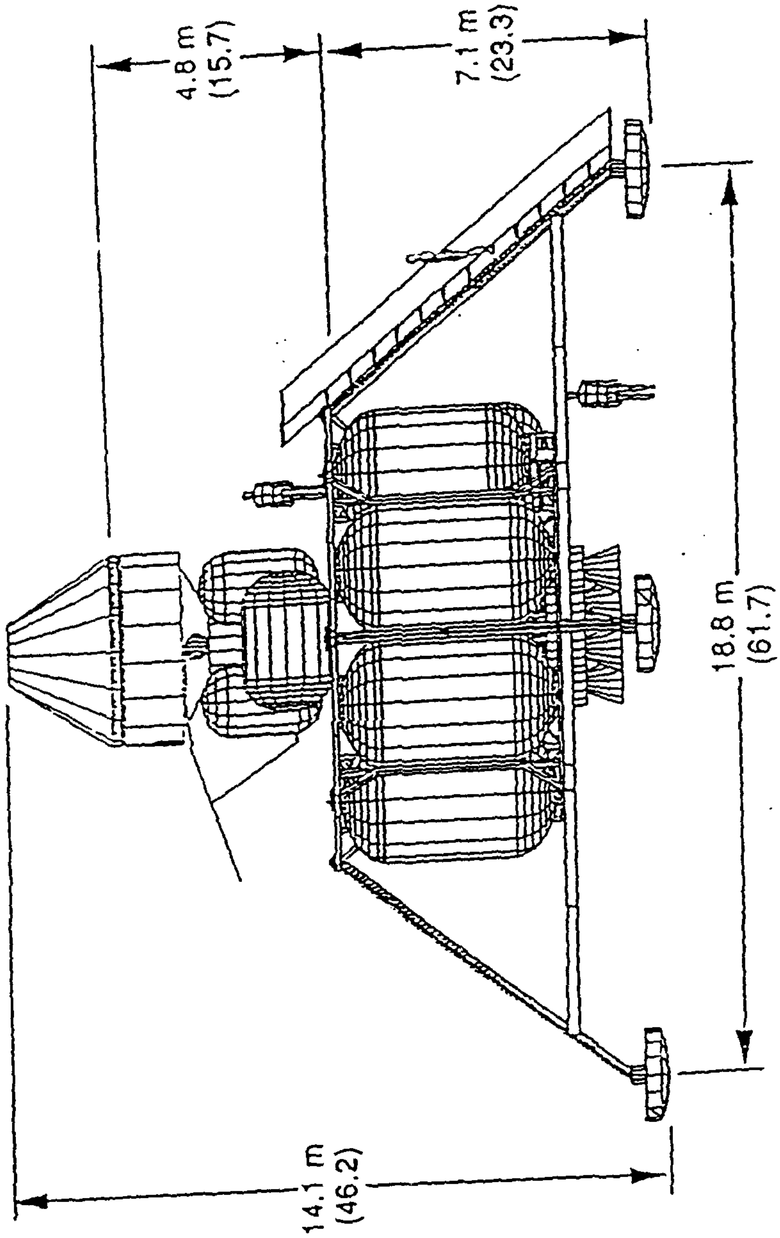
MACH 10-14 TWO-STAGE AND SINGLE- STAGE

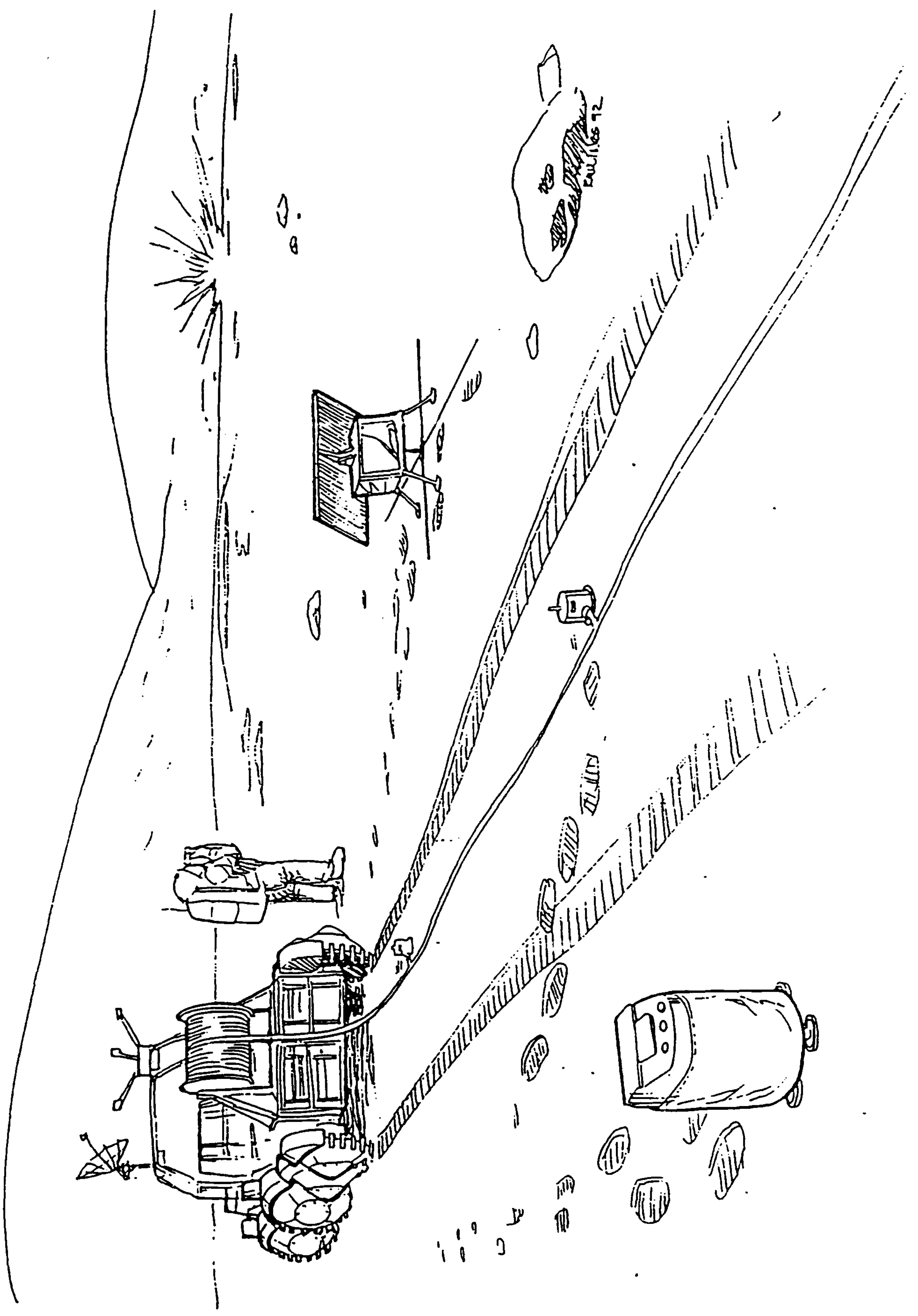

RAWLINGS 92

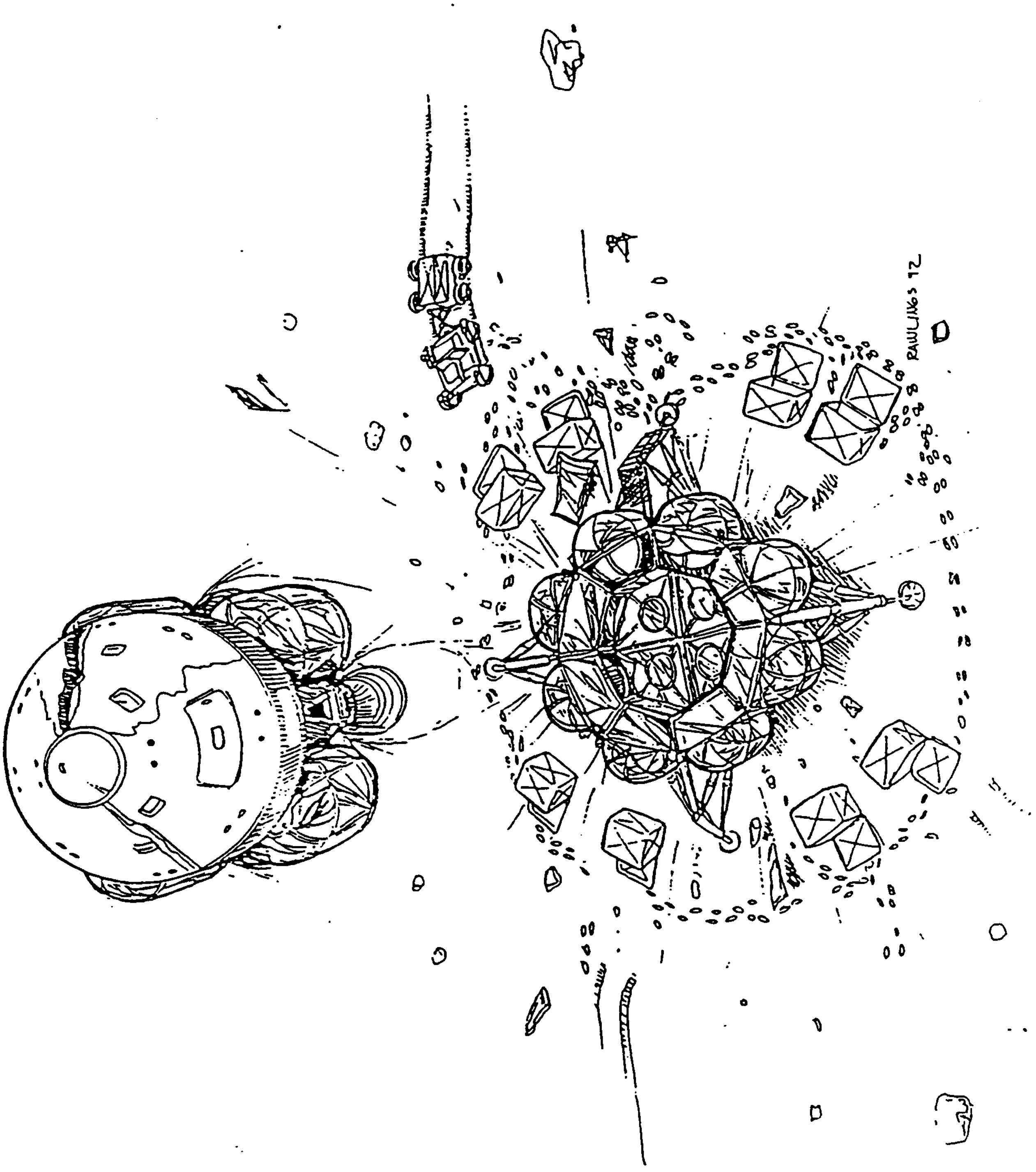

RAWLINGS 77

Lunar Habitat Deployed Configuration




4.8 m
(15.7)
7.1 m
(23.3)
18.8 m
(61.7)
14.1 m
(46.2)



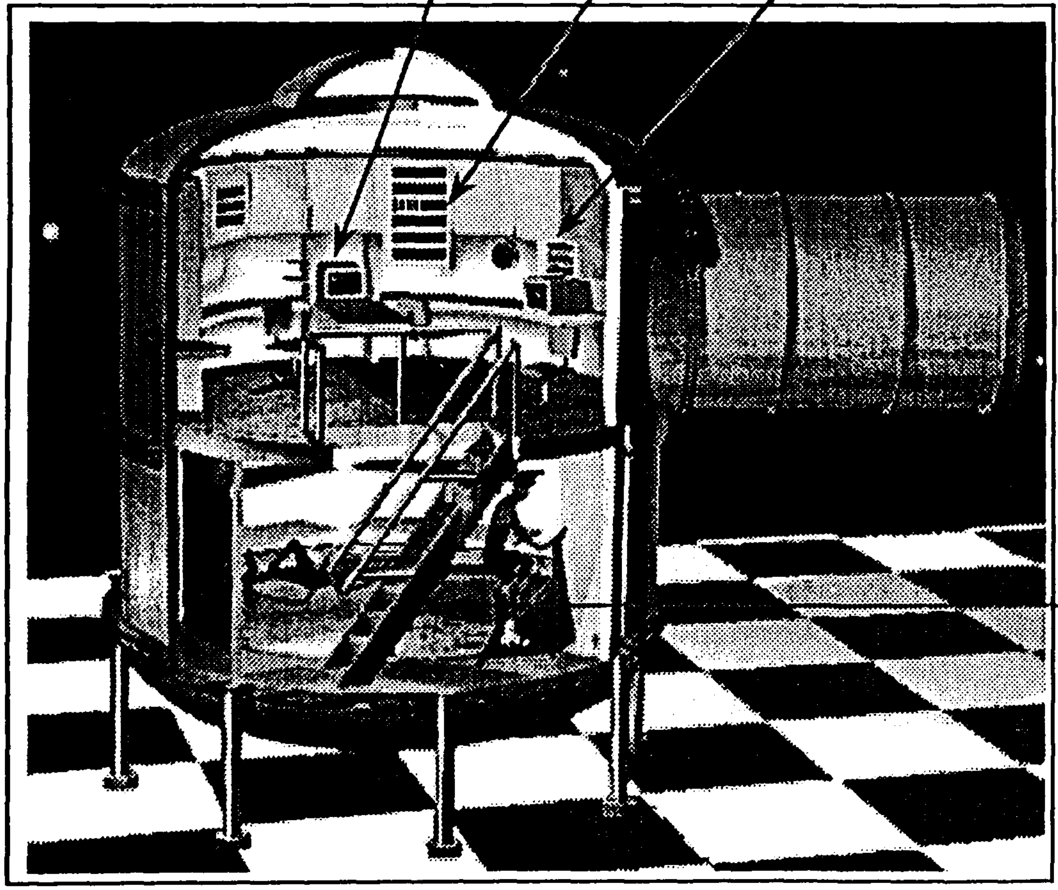


RAWLINGS 92

VIRTUAL REALITY TELEOPERATIONS WORKSTATION

CAUTION AND WARNING SYSTEM

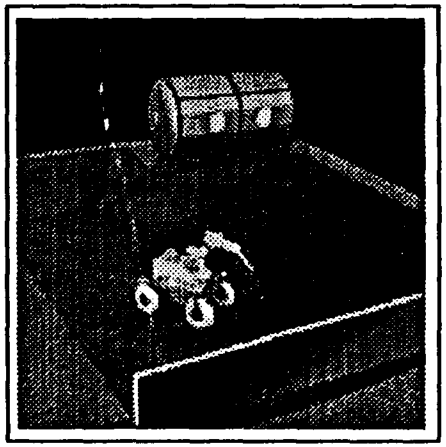
ROBOT CONTROL



CONTROLLED ENVIRONMENT RESEARCH CHAMBER

HUMAN POWERED CENTRIFUGE

LUNAR TERRAIN EXPLORATION SIMULATOR



TELEROBOTIC SYSTEM

HEDP

Goals

- There are four basic goals for the Human Exploration Demonstration Project:

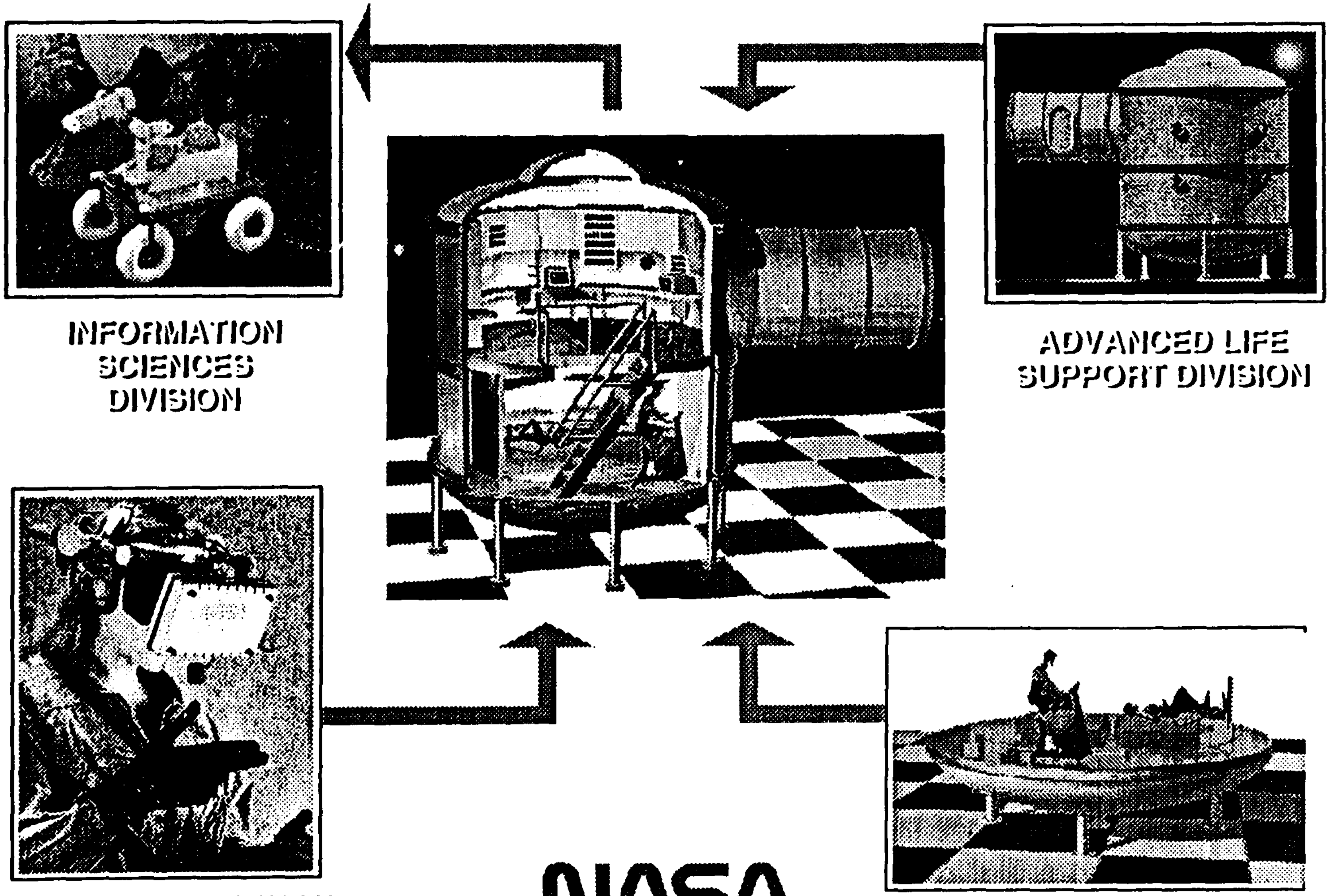
  1. Provide a simulator for the evaluation of technology in an integrated setting
  2. Create a realistic environment for the introduction of new technologies
  3. Enhance the technology development and evaluation process through the synergistic cooperation of multiple Ames divisions
  4. Identify promising technology concepts tp programmatic Centers for new and existing NASA projects

- The HEDP Project will work with the aerospace community and invite them to participate

HEDP

Human Exploration Demonstration Project

INTEGRATED HEDP SYSTEMS



INFORMATION SCIENCES DIVISION

ADVANCED LIFE SUPPORT DIVISION

AEROSPACE HUMAN FACTORS DIVISION

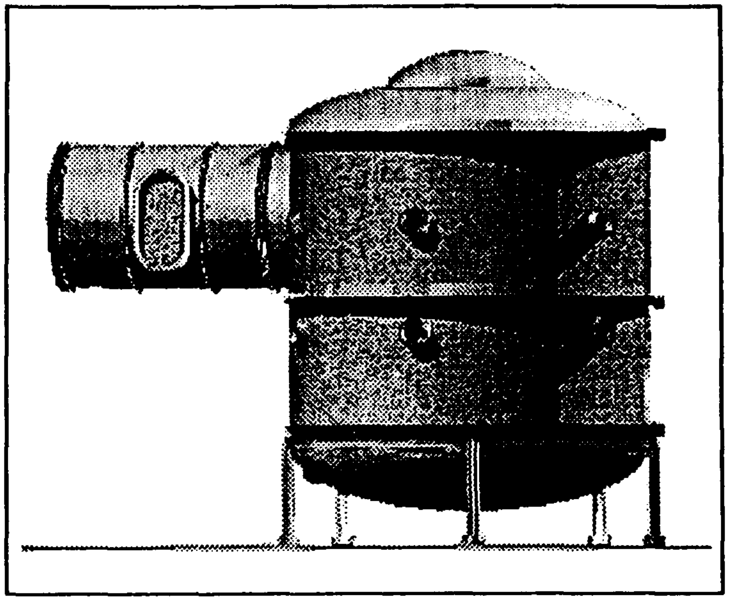
LIFE SCIENCE DIVISION

NASA

Ames Research Center

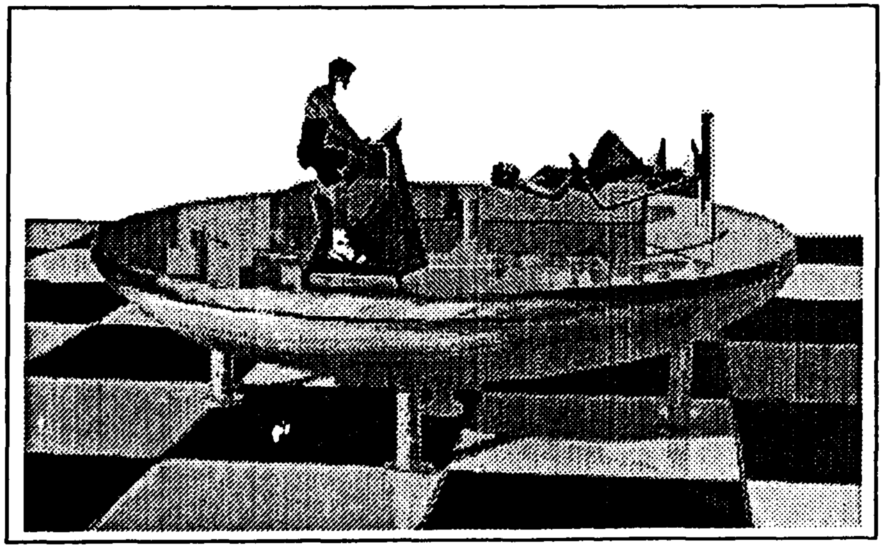
LIVING ENVIRONMENT

- PHYSIOLOGICAL TESTING
- LIFE SUPPORT



LIFE SCIENCE DIVISION

ADVANCED LIFE SUPPORT DIVISION

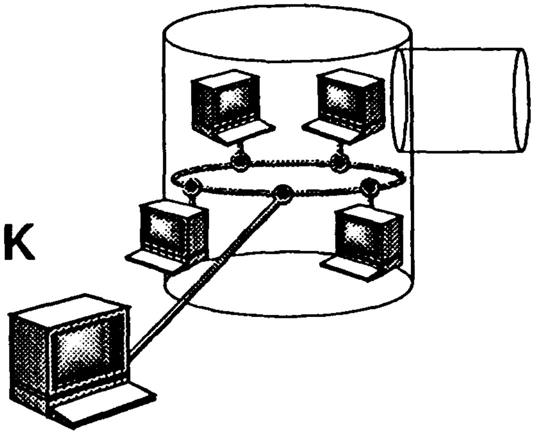


HEDP

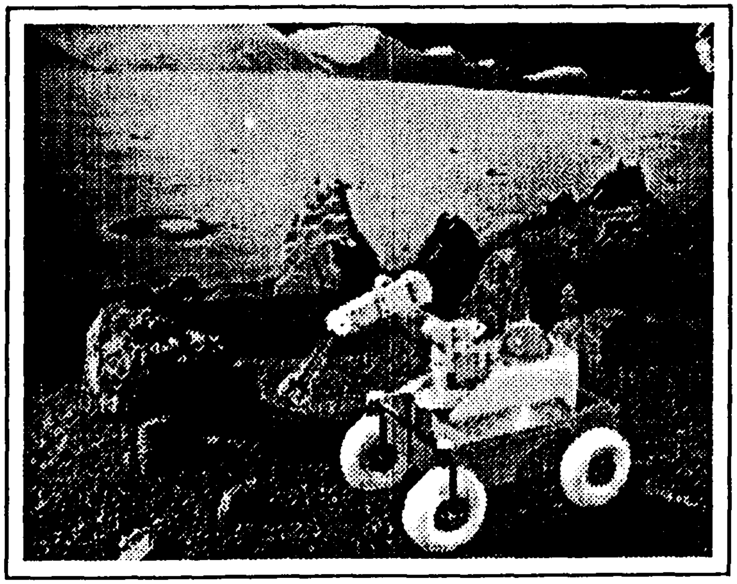
WORKING ENVIRONMENT

AEROSPACE HUMAN FACTORS DIVISION

- TELEPRESENCE
- TELEOPERATION
- FIBER OPTIC NETWORK



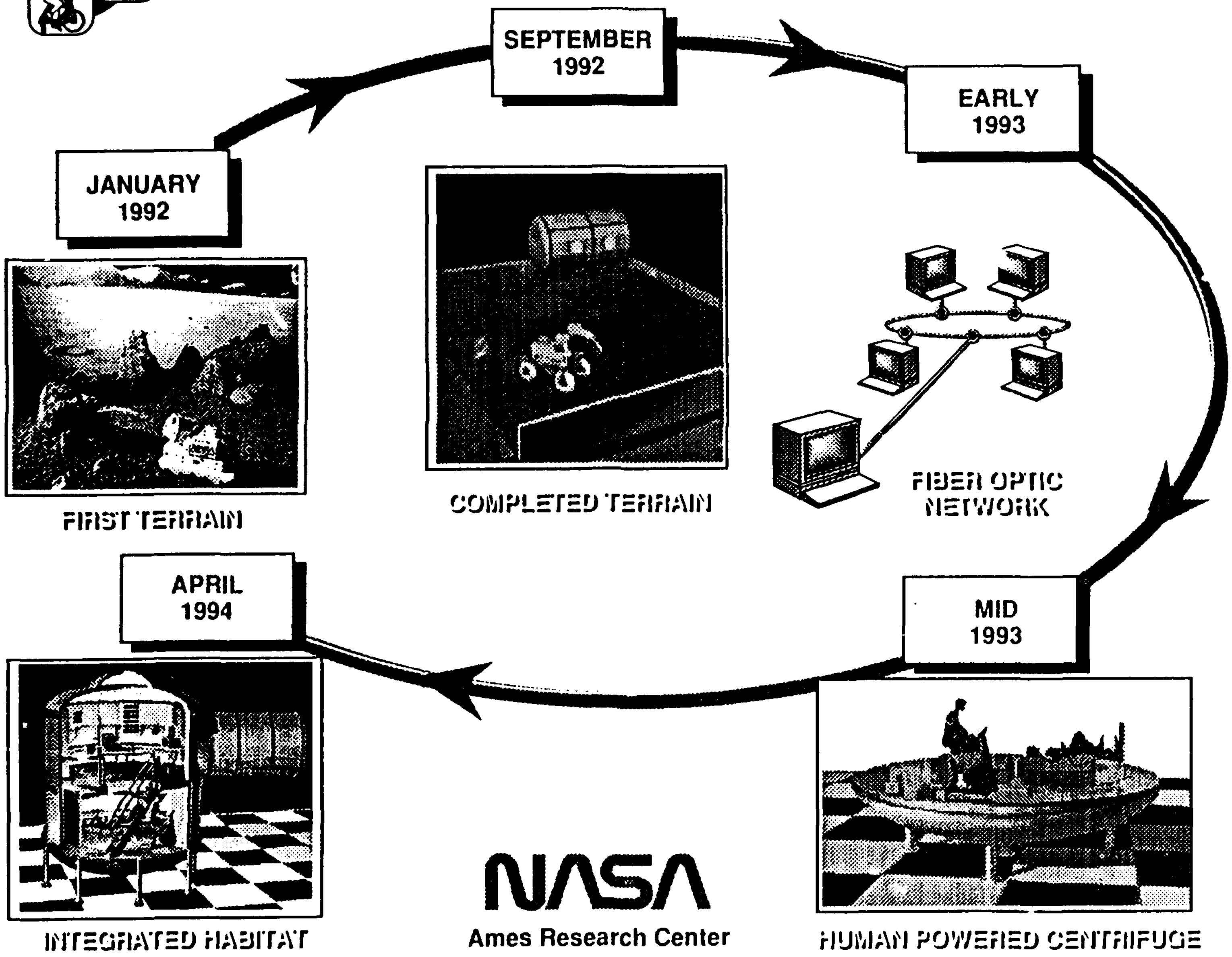
INFORMATION SCIENCES DIVISION



HEDP

HEDP

HEDP EVOLUTION



JANUARY 1992

FIRST TERRAIN

SEPTEMBER 1992

COMPLETED TERRAIN

EARLY 1993

FIBER OPTIC NETWORK

MID 1993

HUMAN POWERED CENTRIFUGE

APRIL 1994

INTEGRATED HABITAT

NASA

Ames Research Center

The Role of Formal Methods in the Design of Complex Real-Time Systems

Armen Gabrielian

UniView Systems

1192 Elena Privada
Mountain View, CA 94040
Tel./FAX: (415) 968-3476
E-mail: armen@well.sf.ca.us

Agenda

- Background on formal methods
- Dealing with complexity and time
- Hierarchical multi-state (HMS) machines
- Verifying safety properties
- Parallelism, scheduling and formal methods
- Multi-level specification
- The future

A. Gabrielian

Background

- Applications of formal methods
  - Definition of system requirements
  - Specification of system operation
  - Analysis of system requirements and design at various levels of abstraction
- Need for formal methods
  - Can't rely on "engineering judgment" anymore
  - Too expensive to develop & test systems
  - Critical nature of many embedded systems
- Shortcomings: Complexity & esoteric notation

Taxonomy of Formal Systems

Axiomatic vs. Executable Formalisms

ISO “standard” formal description techniques:

SDL, Estelle and LOTOS

Process algebra Z Petri nets

Finite-state machines (FSM’s)

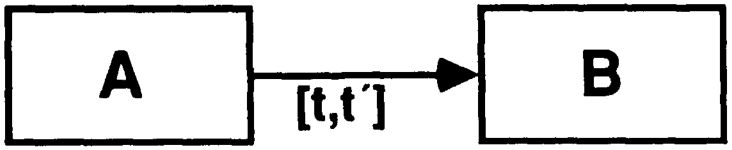
Representing States & Behavior

- Limitation of FSM's
  - State space explosion
  - Limited representation of concurrency
- Shortcomings of Petri nets
  - Difficult to understand
  - Need for numerous "dummy states"
  - "High-level" Petri nets
- Hierarchical state model as a partial solution (best-known example: Statecharts)

A. Gabrielian

Representing Temporal Constraints

- Source of temporal constraints: Physical limitations based on laws of nature, human factors or behavior of subsystems.
- Characteristic of *all* standard formalisms:



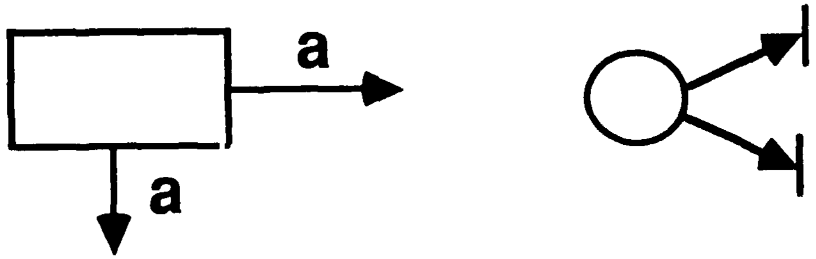
t = Min delay

t´ = Max delay

- Two problems:
  - Assumptions about *future* behavior of system and environment
  - Strictly *self-referential* view of time delays

Views of Nondeterminism

- Nondeterminism arising out of structural considerations:



- Nondeterminism represents:
  - Absence of complete knowledge
  - Specification of a *class* of behaviors
- Not representable in terms of probabilities
- Need to represent temporal uncertainty

A. Gabrielian

HMS Machines

"Hierarchical Multi-State" Machines

State Model: Concurrent & Hierarchical States

- Various models of concurrent behavior
- Transition enablement based on history of states
- Recursive hierarchies
- Multiple clock rates
- Discrete or continuous time

Algebraic Data Model

Temporal Interval Logic (TIL)

Visual Notation

Multiple Verification Methods

TIL = A Language for Time

- O(t) = *At* time t
- [t,t’] = *Always* from t to t’
- <t,t’> = *Sometime* from t to t’
- <t,t’>! = *Sometime-Change* from t to t’
- Examples:
  - [-3,0]“Brake Pressed” ⇒ O(1)“Car Stopped”
  - <-120,-2>“Message received” ⇒ “Data Avail.”

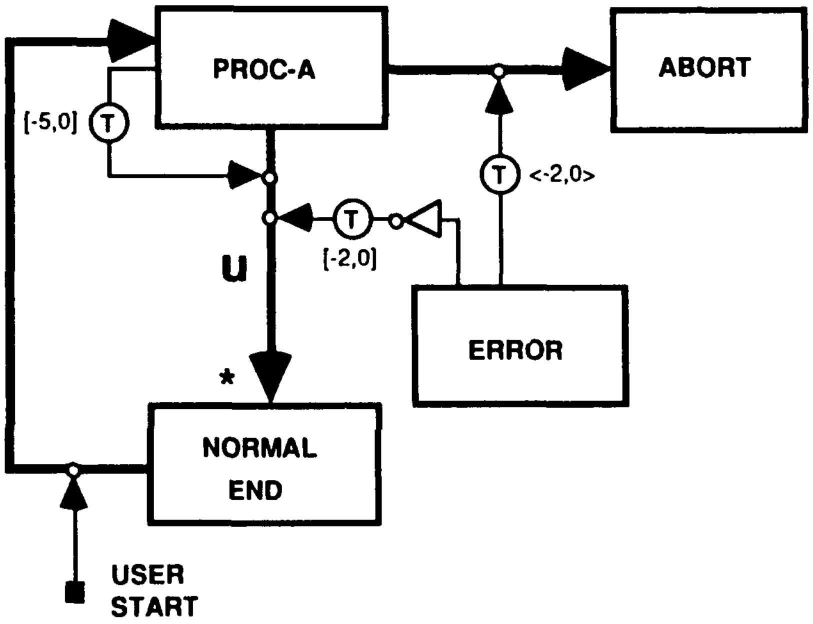
A. Gabrielian

Characteristics of HMS Machines

- Multiple states active ⇒ N vs. 2^N states
- Representation of causality:
  - No assumptions about future events
  - Events depend only on current active states and *past history*, represented by TIL
- Temporal constraints: The constraint on a transition from a state A may depend a delay in state B
- Transitions explicitly designated as deterministic or nondeterministic. For simulation, probabilities may be associated with transitions.

A. Gabrielian

A Simple HMS Machine

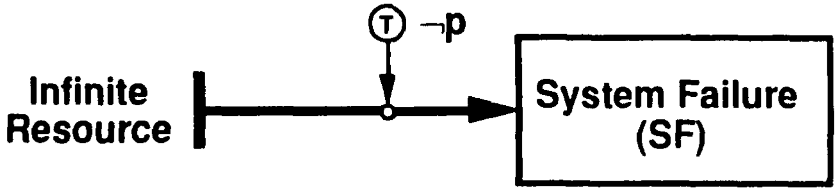


- O(t)u ⇒ [-5,0]PROC-A ∧ [-2,0]~ERROR
- If u is enabled, it ***may*** fire.
- TIL can be used to ***describe*** behavior as well as ***constrain*** behavior

A. Gabrielian

Verifying Correctness

- Types of properties to be verified:
  - Statistical behavior: use simulation
  - Liveness and fairness: not relevant for hard real-time systems
  - Safety: General form of behavioral invariance, expressible as □p (*always* p), where p is a formula in TIL.



- ***Any*** safety property can be represented by a "system failure (SF)" state:

  Property violated ⇔ SF = True

Verification Strategy and Methods

- Overall strategy: Demonstrate that the system failure state (SF) is not reachable.
- Verification methods:
    - Correctness-preserving transformations
    - Model checking
    - Theorem proving
    - Scheduling-based verification
- Testing: Requirements-based test generation
- Interactive simulation

A. Gabrielian

Parallel and Distributed Systems

- Representing parallelism:
  - Multiple copies of specifications: O.K. for simple systems
  - Variables associated with states: Degree of parallelism must be known in advance. Problems with global variables.
  - Recursion: Elegant but difficult to understand
- Representing distributed systems:
  - Asychronous and drifting clocks too complicated to analyze
  - Reasonable assumption: local synchronous clocks with uncertain communication delays

A. Gabrielian

The Future

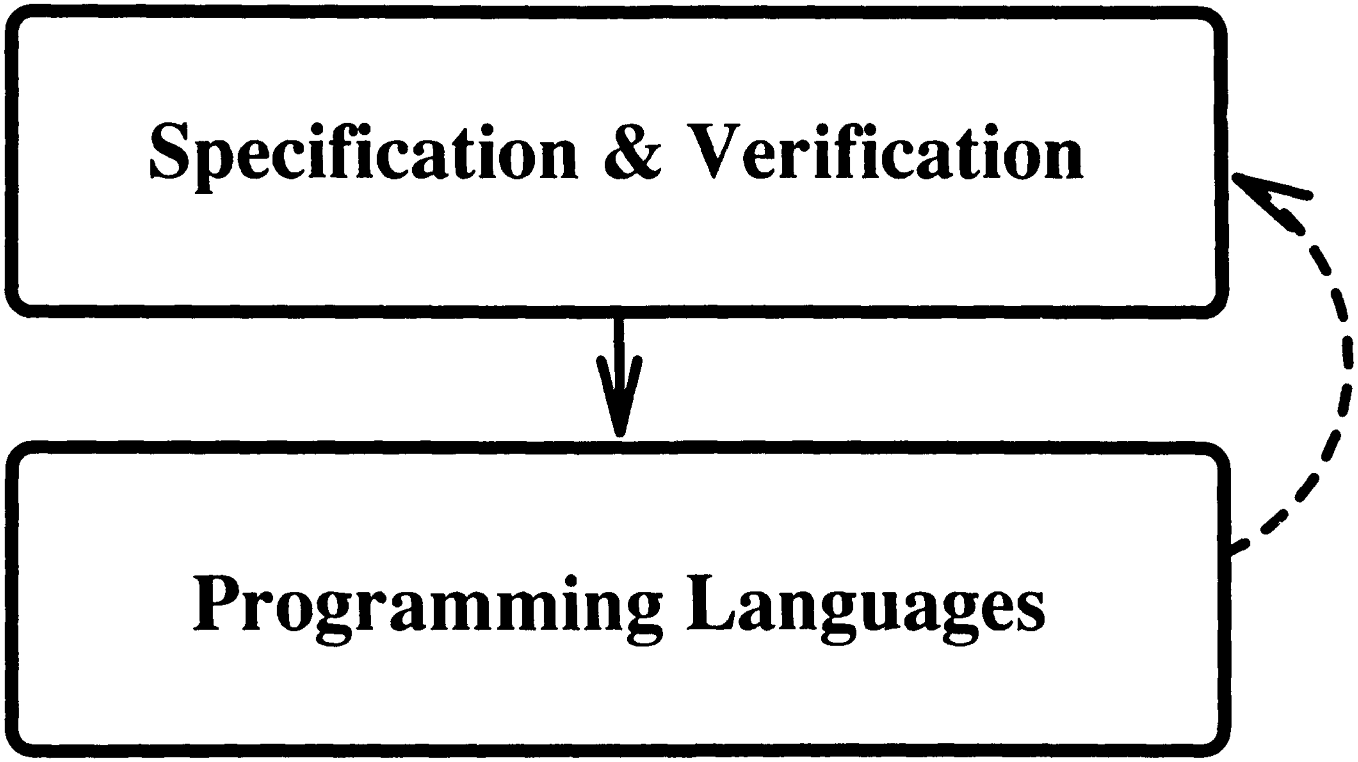
- Formal methcds can help
  - Clarify requirements
  - Verify the correctness of specifications
  - Evaluate designs and implementation
  - Provide means of communication
  - Reduce cost
- Challenges:
  - "Humanized notation" and interfaces with other tools
  - Dealing with complexity
  - Automation

A. Gabrielian

Unifying Real-Time Design and Implementation

Richard Gerber
Systems Design and Analysis Group
Department of Computer Science
University of Maryland
College Park, MD 20742

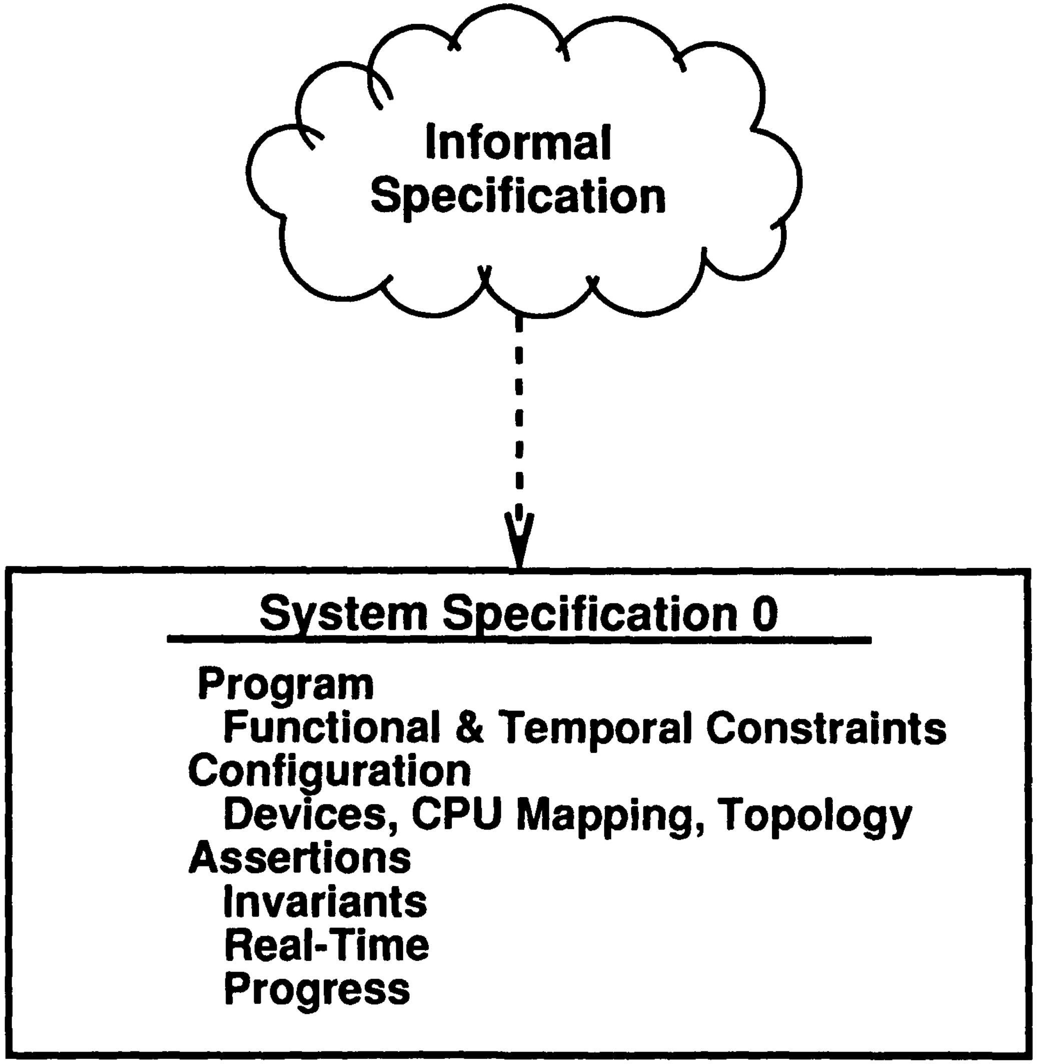
Outline



Specification & Verification

Programming Languages

Specification & Verification



Informal Specification

System Specification 0

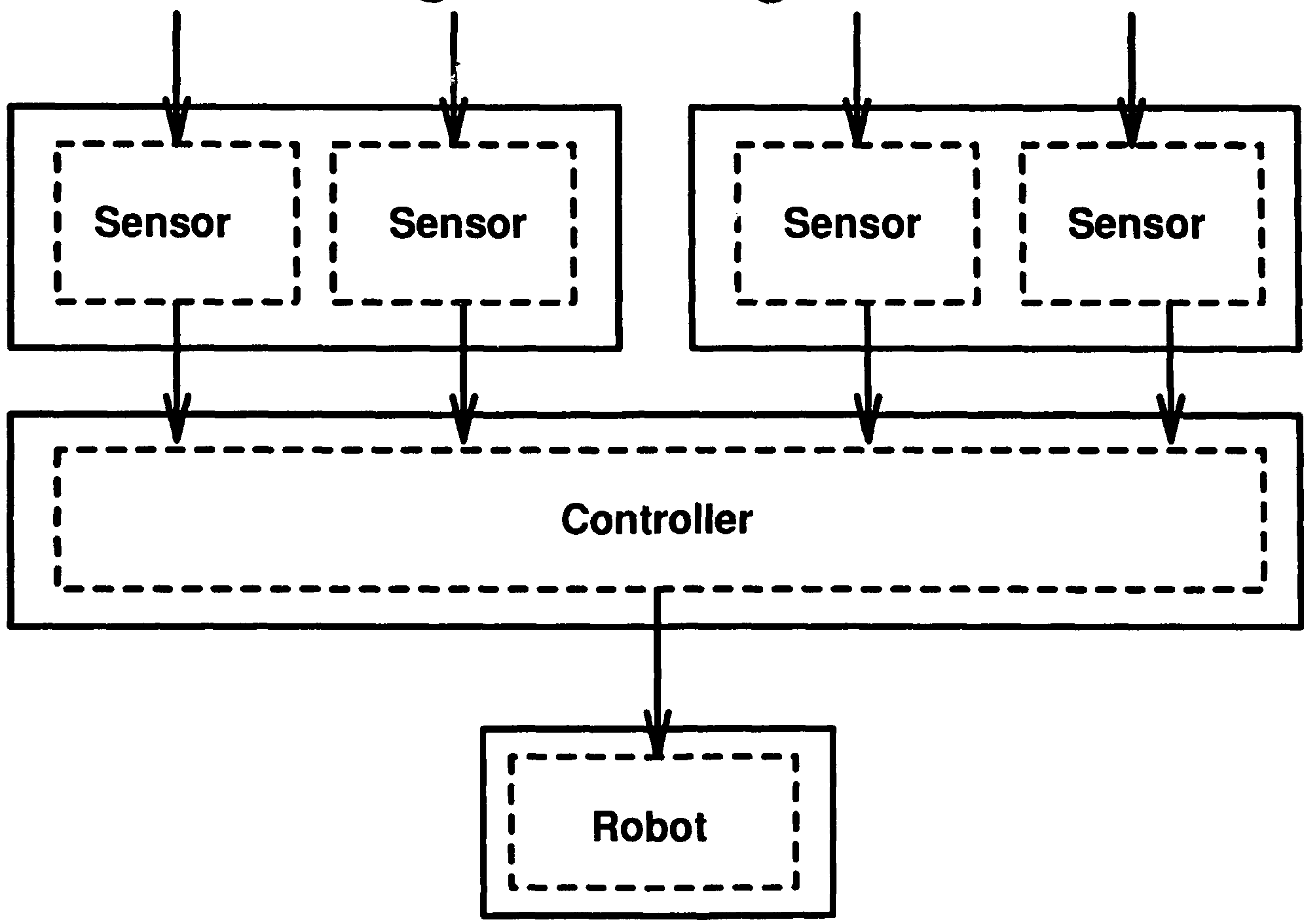
- Program
  - Functional & Temporal Constraints
- Configuration
  - Devices, CPU Mapping, Topology
- Assertions
  - Invariants
  - Real-Time
  - Progress

```
process Sensor(i): 1 ≤ i ≤ 4
  every 6msec
    do
      sample();
    finish within 2msec
      send(ch[i]);
  end

process Controller
  loop
    cobegin
      recv(ch[1]) || recv(ch[2]) || recv(ch[3]) || recv(ch[4])
    coend
    combine();
    send(cmdchan);
  end

process Robot
   every 11msec
     recv(cmdchan);
   end
```

Program Configuration



+ Static Priorities

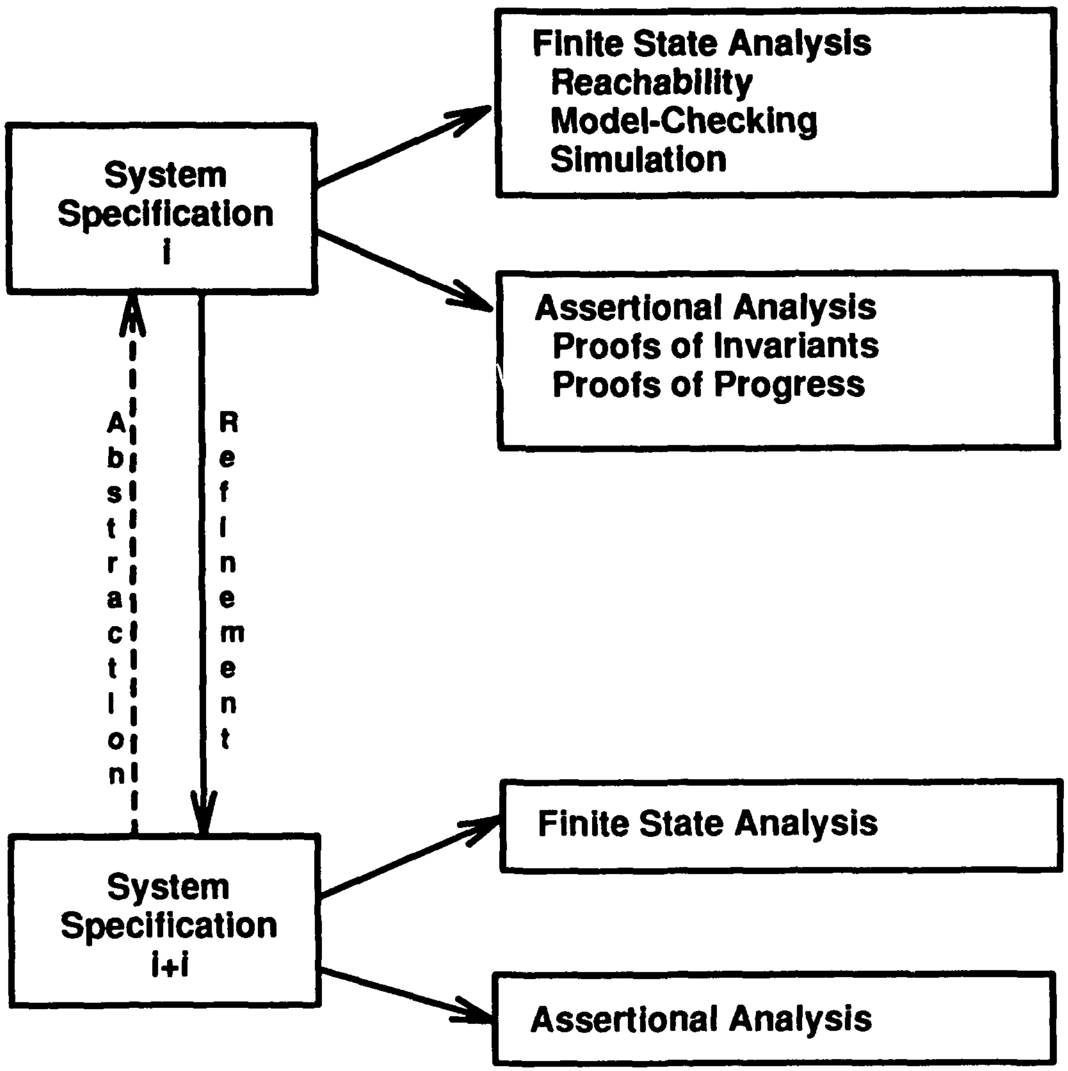
Assertions

- Invariant: $Invariant(\neg error)$.
- Real-time:

  $time = t \land c[1] = .5$ leads_to $time < t + 3msec \land z =$ “stop”
- Progress: $Initial$ leads_to $z =$ “stop”

- **Questions:**
  - Can we formally establish properties w.r.t. the system design?
  - When is the design complete enough to implement?

Verification & Refinement



System Specification i

Finite State Analysis
Reachability
Model-Checking
Simulation

Assertional Analysis
Proofs of Invariants
Proofs of Progress

Abstraction

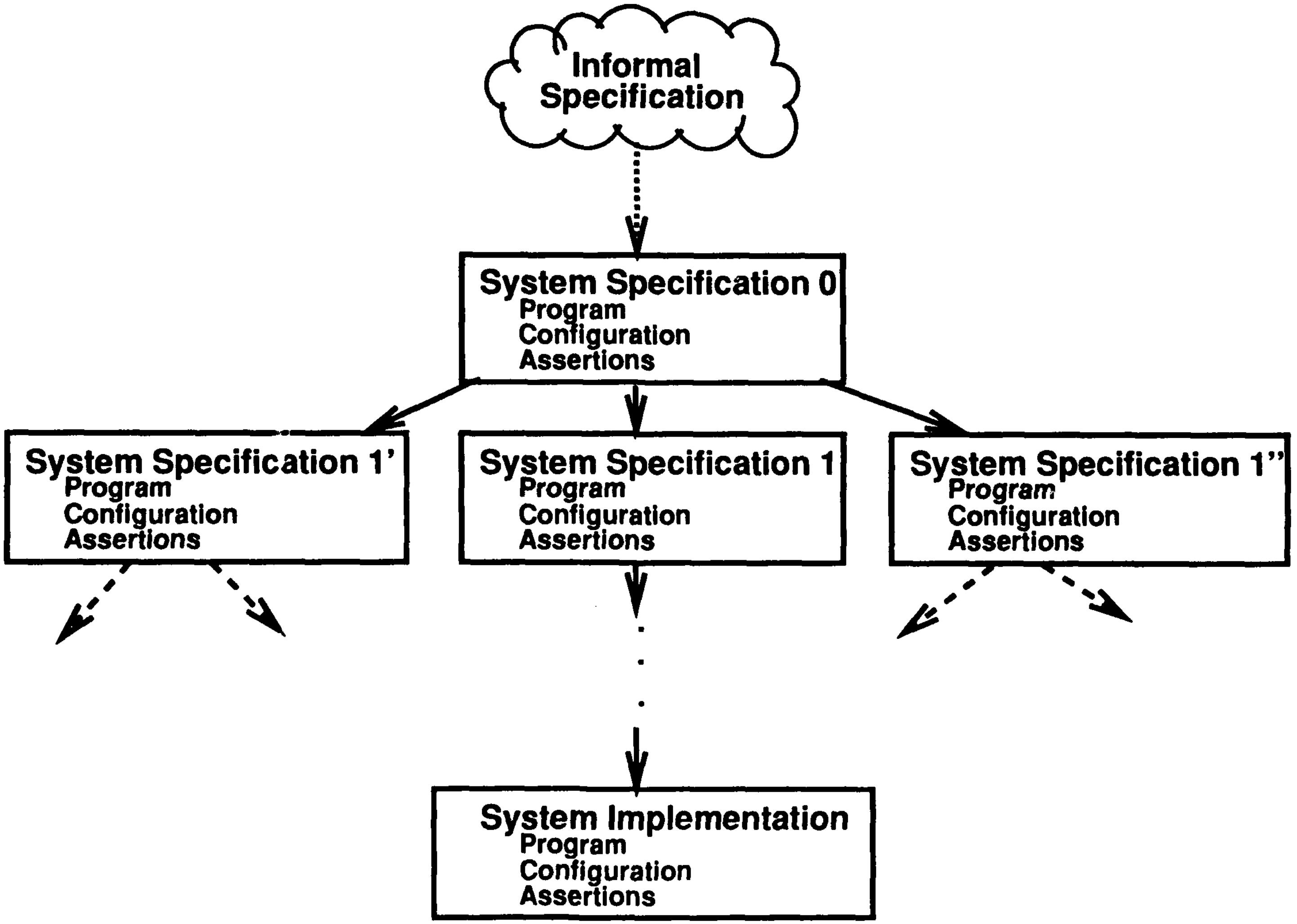
Refinement

System Specification i+i

Finite State Analysis

Assertional Analysis

Goal: Refine to Implementation



Informal Specification

System Specification 0
Program
Configuration
Assertions

System Specification 1'
Program
Configuration
Assertions

System Specification 1
Program
Configuration
Assertions

System Specification 1"
Program
Configuration
Assertions

System Implementation
Program
Configuration
Assertions

Questions on Design Refinement

- Can we formally, logically refine a design into an implementation?
- At which stage does a design language become a programming language?

Research In Programming Languages

language

- Lack of programming methodology for real-time system development.
  - Hand-tuning of code to attain schedulability.
  - Debugging and maintenance problems.
- Lack of production-level programming languages.
  - *Program* correctness is *dependent* on the timing properties of a given hardware platform.
  - Limited expressibility.

TCEL (Time-Constrained Event Language)

Main Goals

- To give a *clean interface* between specification and implementation.
- To provide *machine-independent* description of RT application.
- To provide *consistent code* for RT scheduler.

Approach

An TCEL program is:

- A functional implementation.
- A timing **specification**.

Event-based, Timing Specification

Defines relationship between *events*

- Events are externally observable.

  Examples :

  - stimulus from environment
  - response to environment

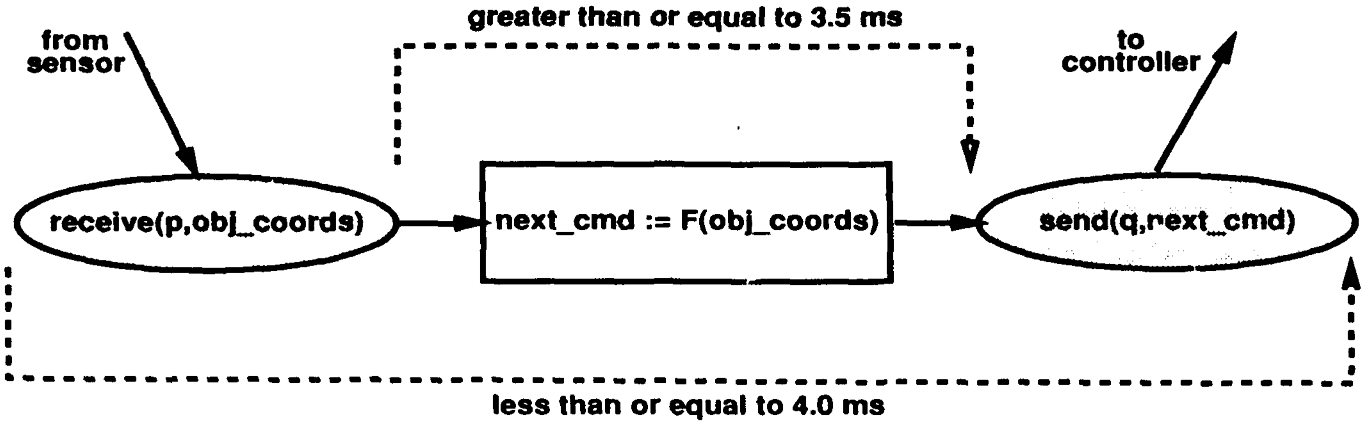
  defined by / derived from specification.

- TCEL events occur instantaneously during the executions of
  - **`send`**(p, v), **`receive`**(p, x).
  - **`outp`**(p, v), **`inp`**(p, x), etc.

- Constraints

  min/*max* delay between events.

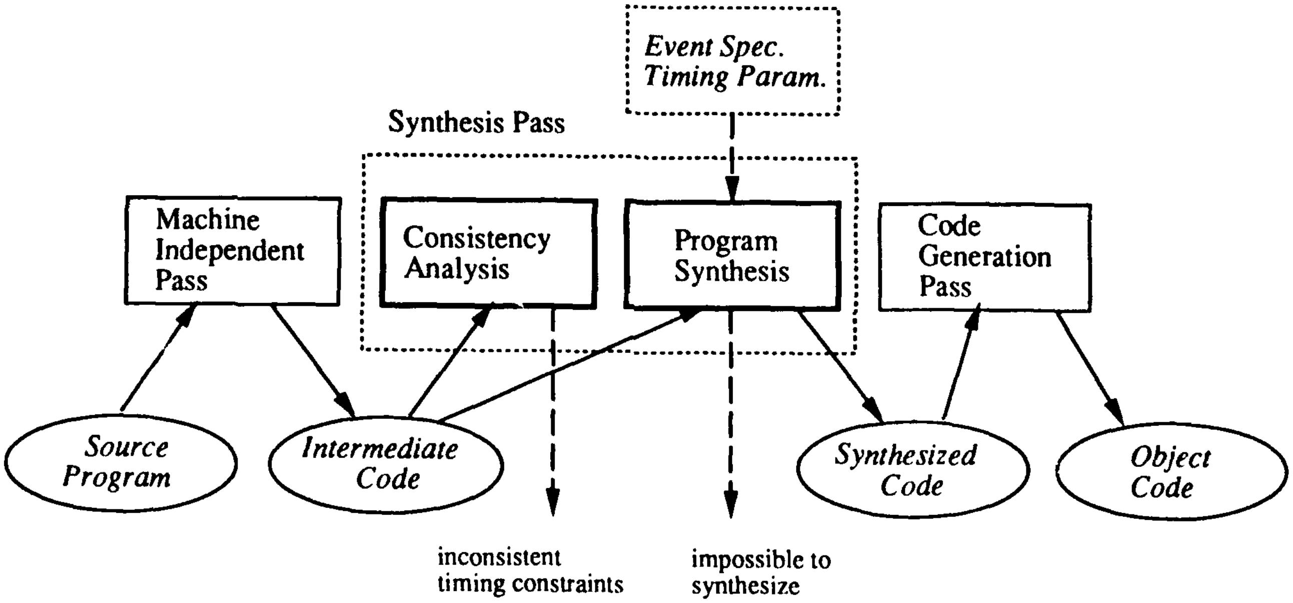
Example


greater than or equal to 3.5 ms
from sensor
to controller
receive(p,obj_coords)
next_cmd := F(obj_coords)
send(q,next_cmd)
less than or equal to 4.0 ms

```
do
  receive(p,obj_coords);
start after 3.5 ms
finish within 4.0 ms
  {
  next_cmd = F(obj_coords);
  send(q,next_cmd);
  }
```

```
do
  {
  receive(p,obj_coords);
  next_cmd = F(obj_coords);
  }
start after 3.5 ms
finish within 4.0 ms
  send(q,next_cmd);
```

Compilation Process

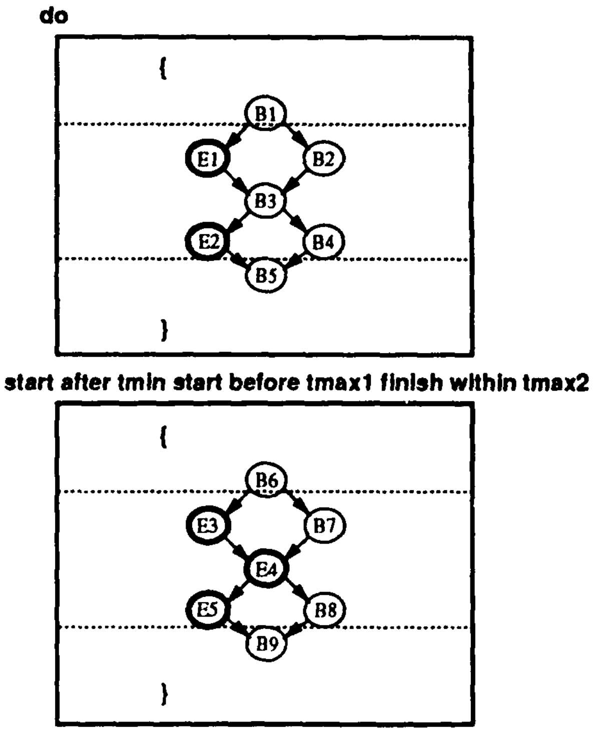


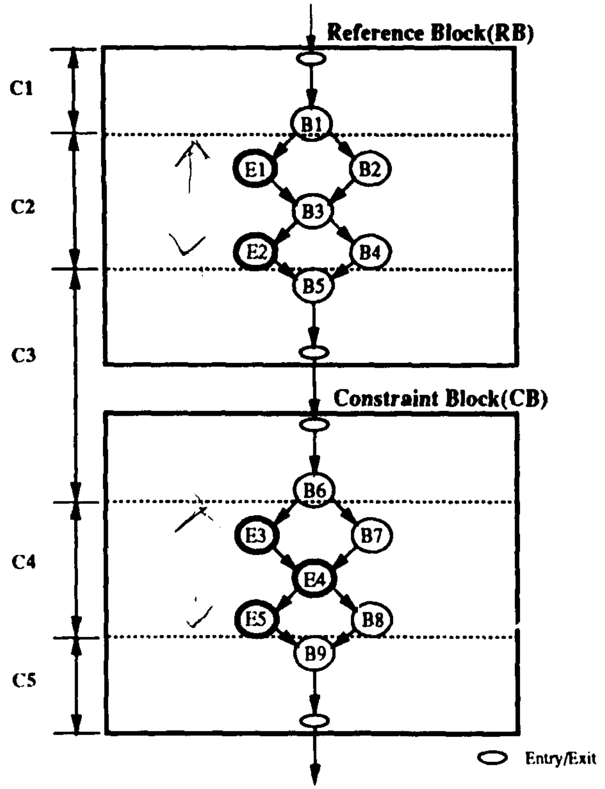
Program Synthesis

- is the process of *rearranging code* to attain *consistency* with respect to given timing constraints.
- is a two-step process:
  1. A timing construct is decomposed into *sections*.
  2. Code scheduling is used to attain *consistency*.

Step 1: Section Generation

- A *section* is a control flow subgraph having a single *entry* node and a single *successor*.





Step 2: Code Scheduling

- If section C is determined to be *inconsistent*, operations in C are moved into another section in an attempt to reduce $wt(C)$.

- *Trace scheduling* is used.

  1. Select trace t in C such that $wt(t)$ violates constraints.

  2. Move operations trace by trace until C becomes consistent.

Example: Specification & Source Program

1. Every 10.0 ms, receive message from sensor.
2. Delay at least 1.5 ms after receiving message.
3. If object was detected, send new commands to arm1 and arm2 within 4.0 ms of receiving message from sensor, and before next message.

```
every 10.0 ms
 do
  receive(Sensor, dim);              [0.4ms]
 start after 1.5 ms
 finish within 4 ms
  if (!null(dim)) {                  [0.22ms]
    z1 = newcmd(dim, loc1);          [1.0ms]
    z2 = newcmd(dim, loc2);          [1.0ms]
    send(arm1, z1);                  [0.4ms]
    send(arm2, z2);                  [0.4ms]
    }
```

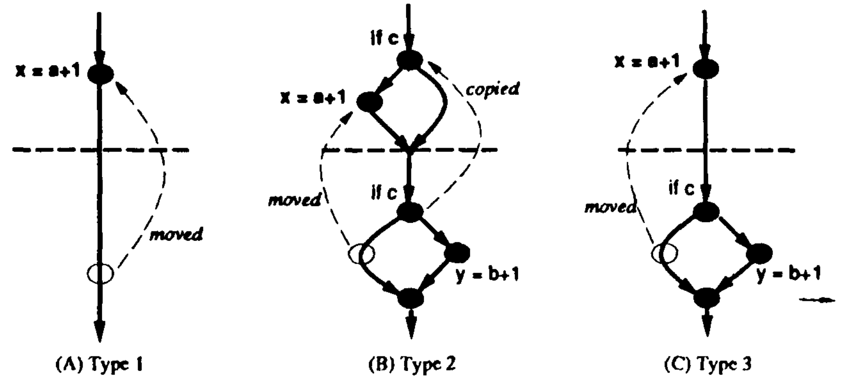
Section Decomposition & Transformation

```
S0: (S0.start[p]≥ p×10ms,
    (S3.finish[p]≤ (p+1)× 10ms)
S1: receive(Sensor, dim);
S2: {  }
S3: (S3.start≥S1.finish+1.5ms,
        S3.finish≤S1.finish+3.6ms)
        {
        c = !null(dim);
        if (c) {
           z1 = newcmd(dim, loc1);
           z2 = newcmd(dim, loc2);
           send(arm1, z1);
           send(arm2, z2);
           }
        }
```

```
S0: (S0.start[p]≥ p×10ms,
    (S3.finish[p]≤ (p+1)× 10ms)
S1: receive(Sensor, dim);
S2: {
    c = !null(dim);
    if (c)
       z1 = newcmd(dim, loc1);
    }
S3: (S3.start≥S1.finish+1.5ms,
       S3.finish≤S1.finish+3.6ms)
       if (c) {
          z2 = newcmd(dim, loc2);
          send(arm1, z1);
          send(arm2, z2);
          }
```

Core Transformations

Type	Critical Section's WCET	Overall ET	Overall WCET
1	reduces	maintains	maintains
2	reduces	maintains	maintains or increases
3	reduces	increases	maintains or increases



(A) Type 1

(B) Type 2

(C) Type 3

Preference : Type 1 > Type 2 > Type 3

Global Transformation Application

- Optimize concurrent tasks to achieve system-wide schedulability.

⟹ Close interactions between the compiler and the scheduler.

Example: Critical zone-driven transformations in Rate-Monotonic Scheduling.

⟹ We decompose & merge tasks to lessen utilization in critical zones.

Conclusions

- An "implementation-driven" approach to specification.
- A "specification-driven" approach to programming languages.

Needs:

- Develop more robust, scalable verification techniques (long-term payoffs).
- Further cement ties between design languages & programming languages (shorter-term payoffs).
- Further cement ties between compilation & scheduling (short-term payoffs).
- But, we need accurate static timing analysis tools!

Large, Distributed, Parallel Architecture Real-Time Systems Workshop

Institute for Defense Analysis
Alexandria Virginia

March 17, 1993

Ada Real-Time Methodology Assessment Lessons Learned at Ames Research Center

Andy Goforth
Advanced Data Systems Architecture Group

Strategic Goals

- To bring advanced information technologies into NASA through
  - —Inhouse R & D efforts to create a center of excellence
  - —Parterships with industry and academia to create a proving ground for technologies that significantly improve existing engineering practices

- To enable NASA to take advantage of competitive technologies

- To provide to industry and to academia
  - —Future mission requirements through scenerios and test environments that identify "tall poles" or "show stoppers", and "cost drivers"
  - —Independent and impartial assessment of their technologies

Overview of ADSA Activities

- Systems Engineering and Analysis (SE&A) for NASA projects
  - — Trade Studies
    - > 386 versus 486-based flight avionics
    - > Dynamic End-to-End DMS modeling
  - — Engineering Evaluations
    - > Ada CIFO stress testing
    - > Programming Guideline Compliance
- Assessment and Research
  - — Evaluation and Integration of DARPA sponsored advanced processing
    - >Iwarp, Multimax (MACH 3.0)
  - — Pathfinding in real-time systems that lack good engineering practice
    - >Real-Time Design Methodology
    - >Ada 9X

Problem Statement

"Current real-time design methodologies can lead to elaborate, highly complex designs that can be shown to be inefficient."

Institute for Defense Analysis
Report HQ 91-037919
Dr. Norman Howes

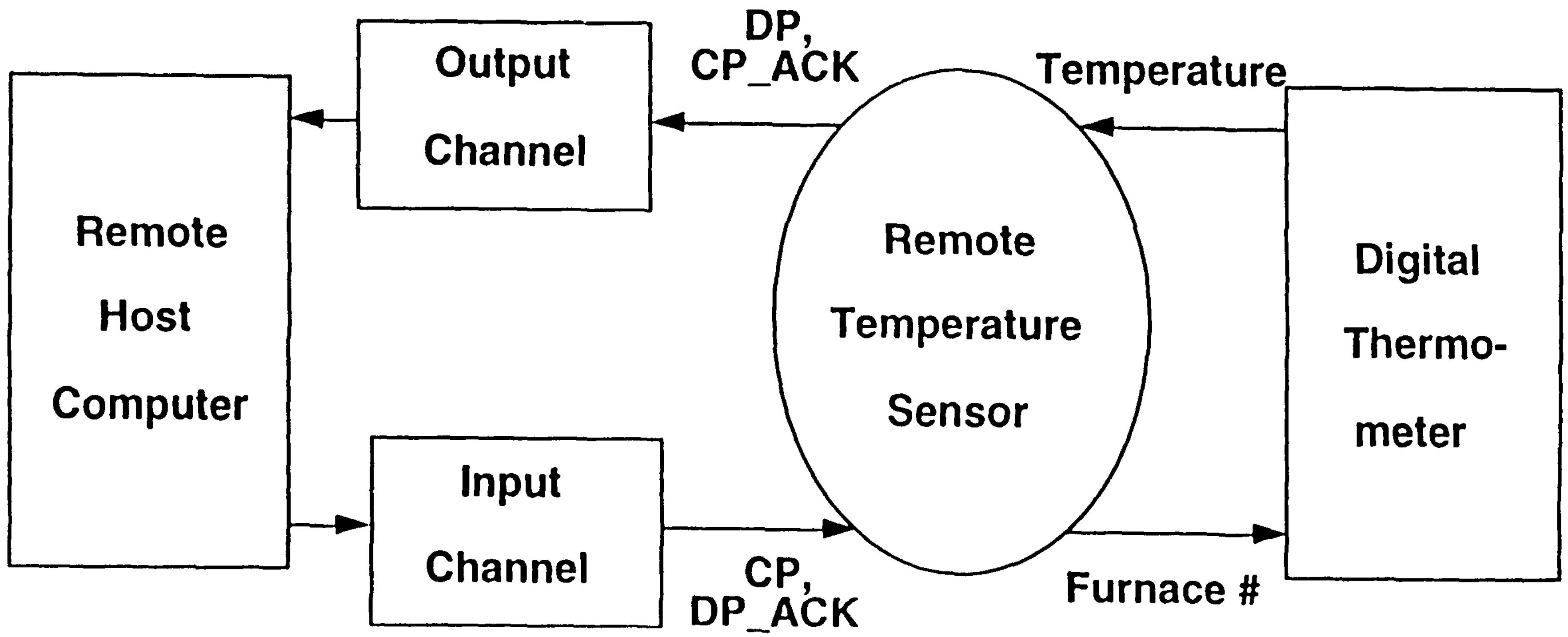
Design Methodology

- *Designing Large Real-Time Systems with Ada* by Kjell Nielsen and Ken Shumate
    - — Text used in universities and industry
    - — Contains Remote Temperature Sensor (RTS) case study which is one of six with references in literature
    - — One of several real-time methodology approaches
    - — Has elements of "Object Oriented Programming" approach

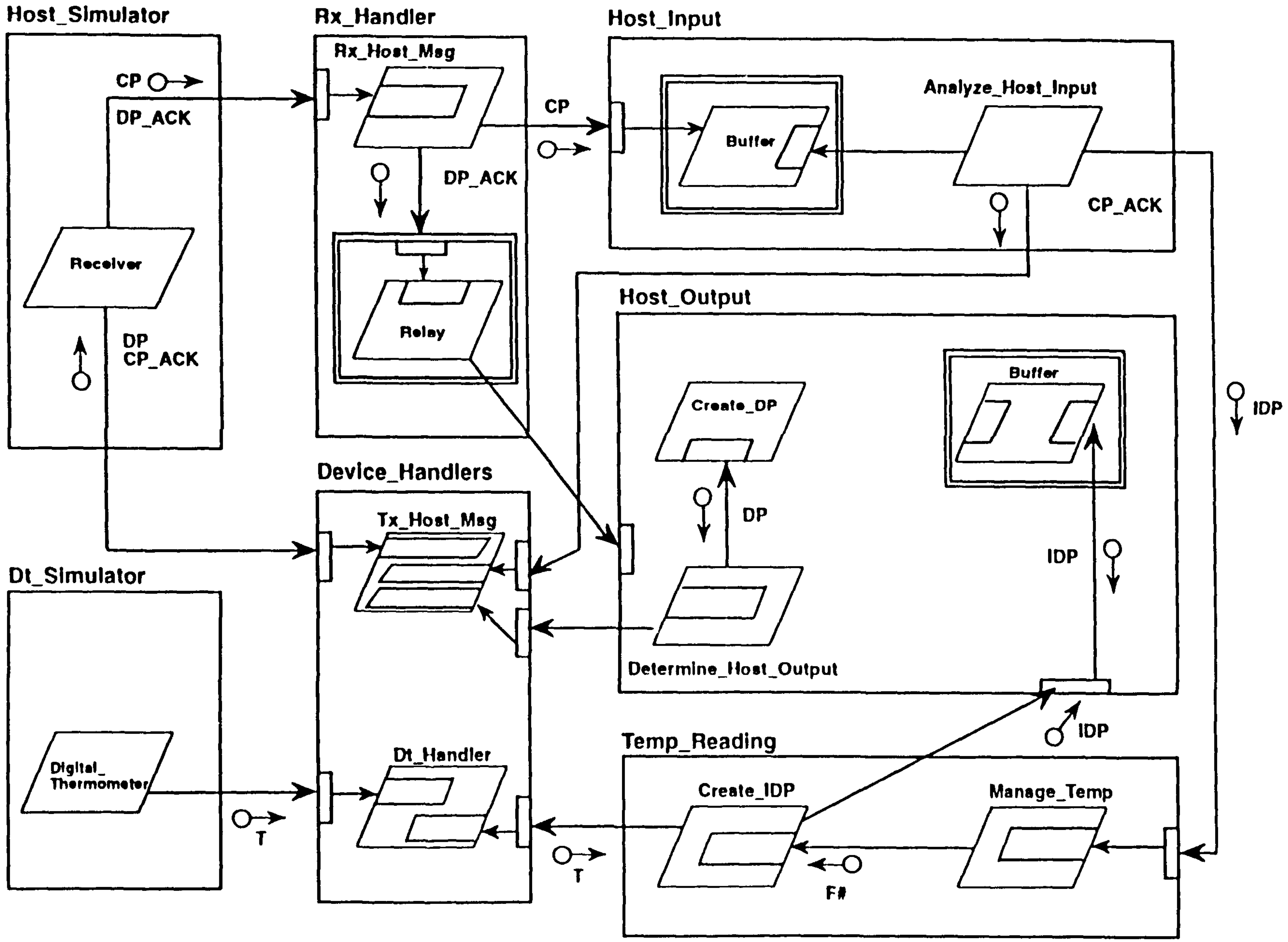
Q4 92 and Q1 93 Accomplishments and Findings

- Established common real-time test case with IDA
- Benchmarked and evaluated two different implementations of test case that are based on different methodologies
- Found design flaw in one of the implementations that made the test case not work on a multiprocessor
- Confirmed IDA's findings on the two different test cases and identified another performance issue

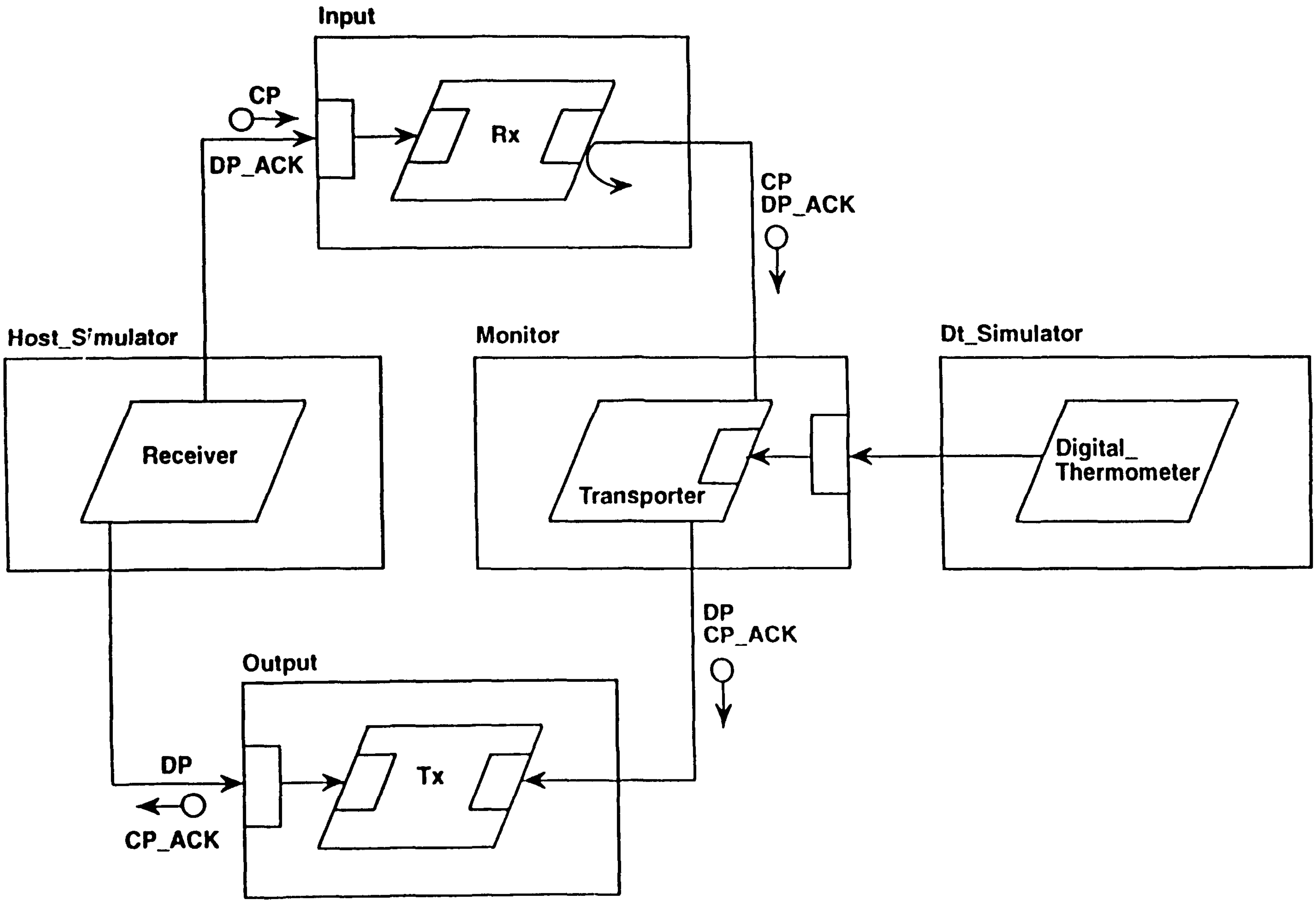
Overview of Remote Temperature Sensor (RTS) Case Study



Original Nielsen and Shumate Design of Remote Temperature Sensor

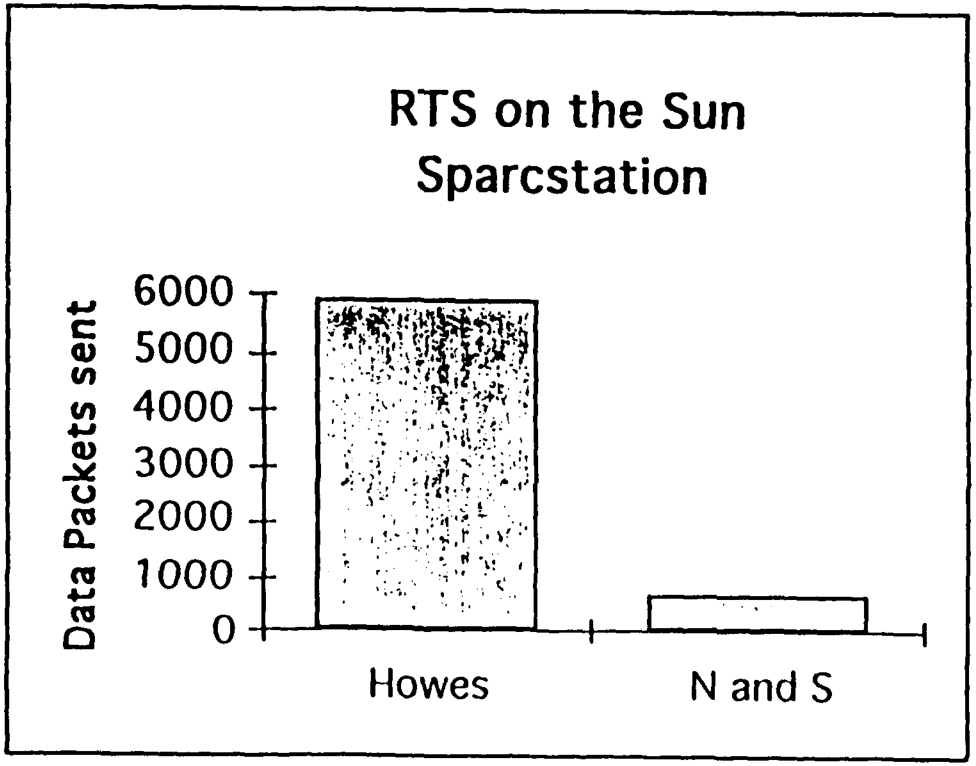


Redesigned Remote Temperature Sensor



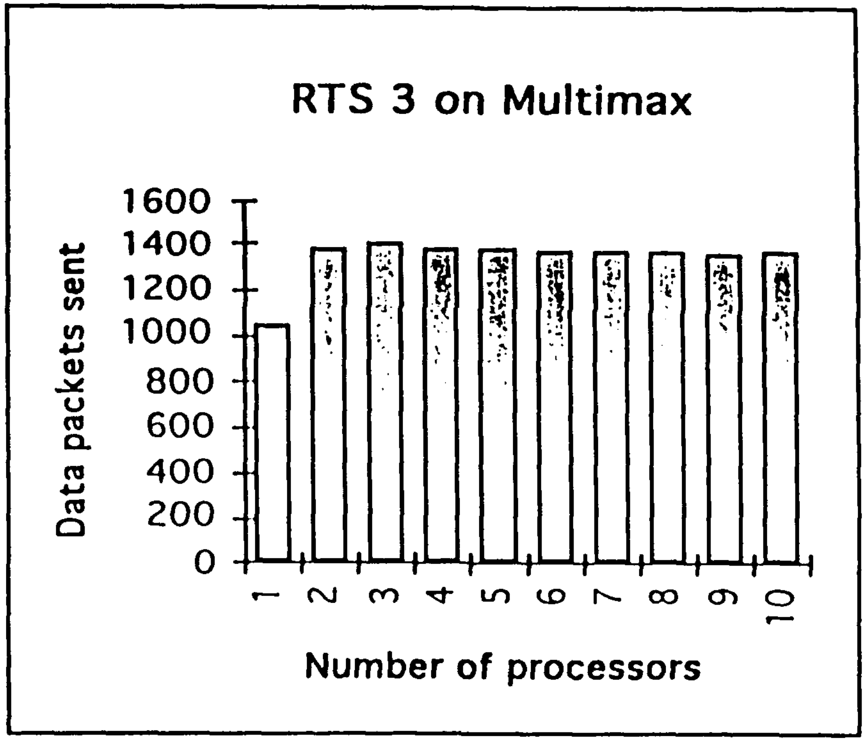
Redesigned RTS Throughput

On the SUN Sparcstation, Howes' redesign of RTS had higher throughput than the original Nielsen and Shumate design.



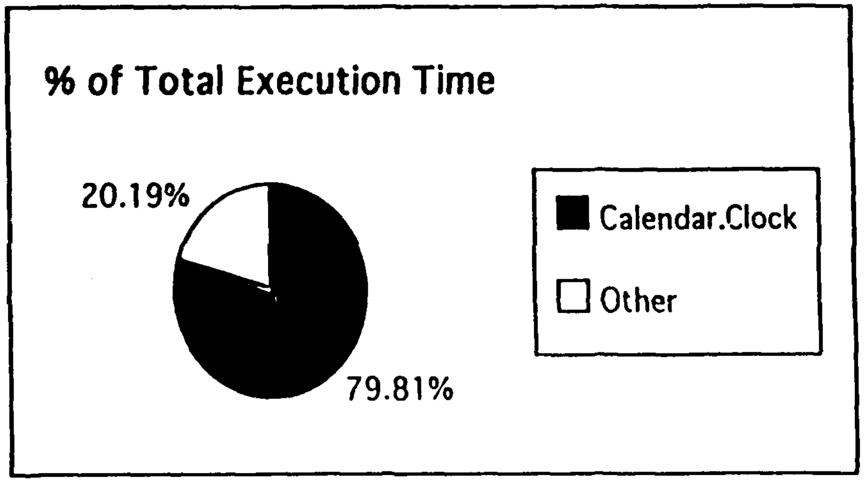
IDA's RTS Throughput on Multprocessor

Beyond two processors, we do not measure an increase in throughput for Howes' redesign of RTS.



Finding with Howes' Redesign

- Profile report shows almost 80% of time was spent in Calendar.Clock.



- An *Either Or* issue of
  - Slow Clock implementation
  - Excesεive calling due to program design

Findings with the Nielsen and Shumate Design

- Unable to run it consistently on Encore Multimax with more than 1 cpu configured
- Performance is comparatively slow with 1 cpu configuration on Multimax

Reason:

- N & S scheduling depends on cpu performance and completely fails when performance goes below critical threshold
- Over head from task interaction is significant in the Nielsen and Shumate design—profiler on Sparc shows about 85% of execution time spent in rendezvous

Unsafe Timing in Task Manage_Temperature_Reading

```
loop
   select
      when ((Check_Temp_Time - Clock  ) > 0.0 ) =>
         accept

         -- accept a Control Packet

      or
         delay (Check_Temp_Time - Clock  );

        ⇕   - - Time to execute statements here ↕ depend on CPU performance

         Check_Temp_Time := Check_Temp_Time + Read_Temp_Period;

   end select;
end loop;
```

Assessment to date

- Nielsen and Shumate's design of RTS is
  - — unnecessarily complicated in our opinion because it took a considerable amount of effort just to get it to work (difficult to analyze)
  - — relatively slow performance and difficult to improve
  - — concurrency does not work well on multiprocessor
  - — sensitive to performance of rendezvous on Sparc platform

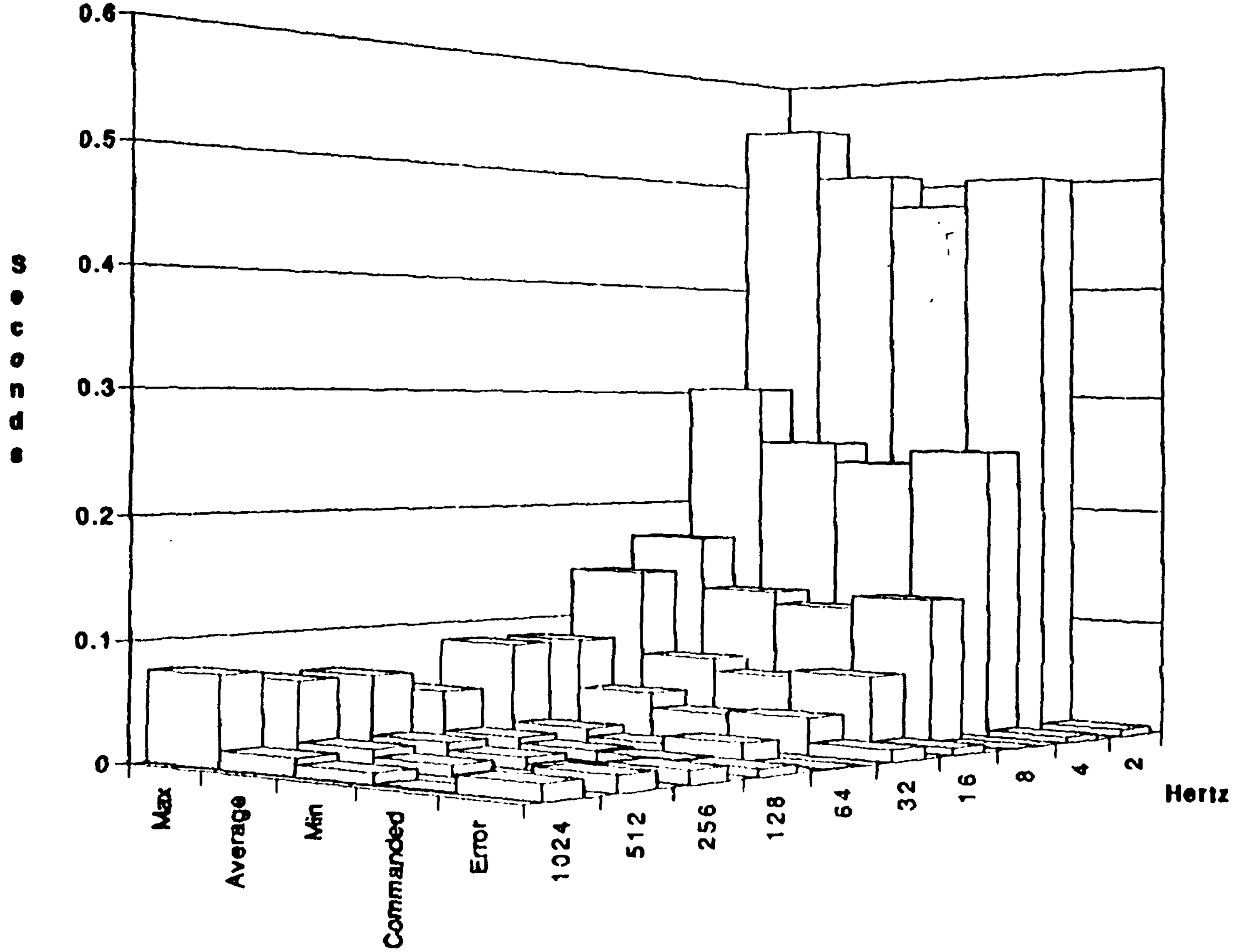
- Howes' redesign of RTS is
  - — relatively less complicated
  - — more efficient
  - — runs on a multiprocessor (tested on up to 14 cpus)
  - — sensitive to the performance of Calendar.Clock function on Sparc platform

- Howes' test case scenerio—currently a stress test that represents bursty demand and relatively higher rate of sensor reading than discussed in Nielsen and Shumate text

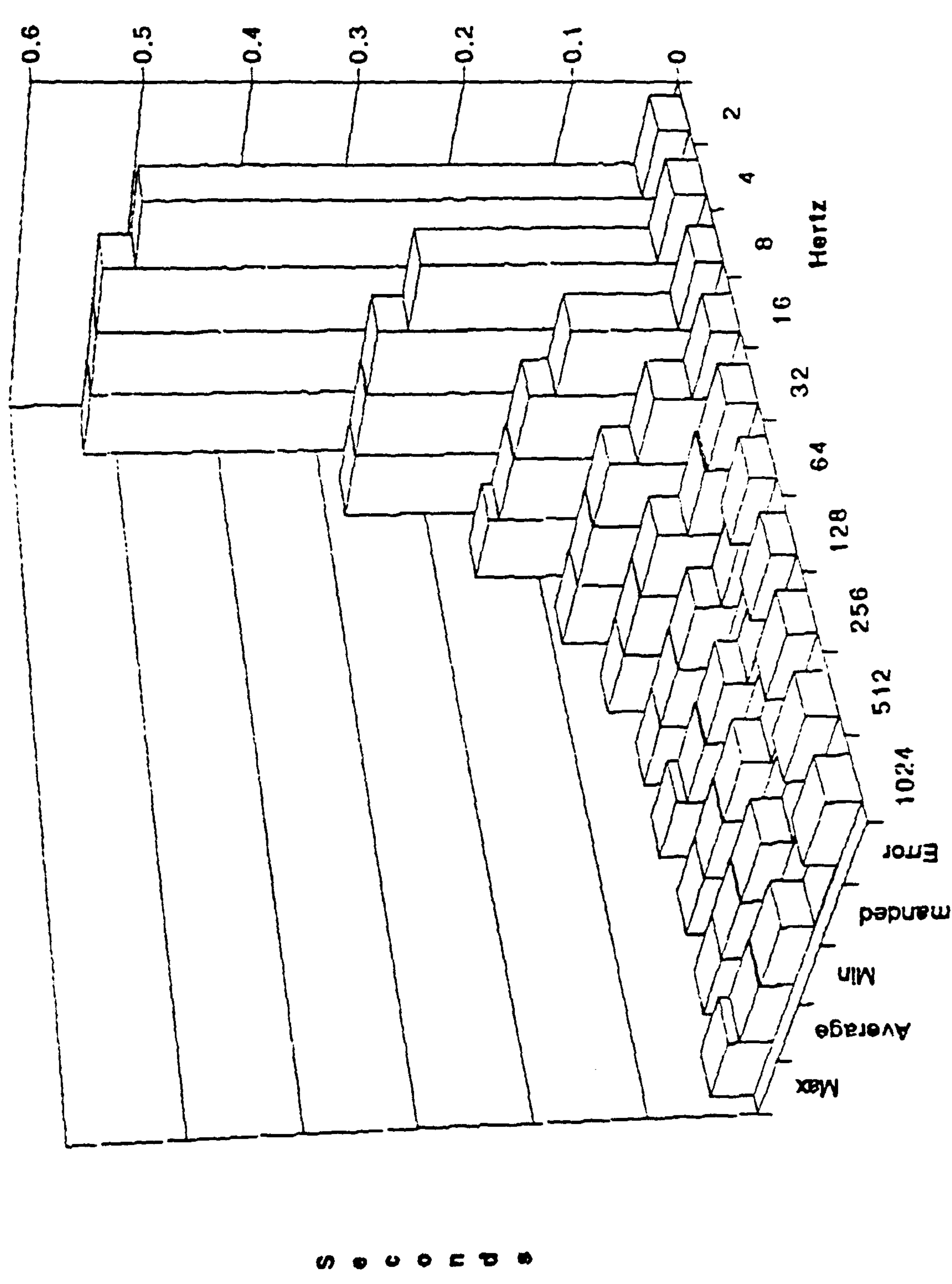
Real Time Design Methodology Assessment

- Flawed with software / hardware dicotomy
- No consideration of "real world" time critical functions and services
- Example of "real world", "nitty gritty" time related functions, namely
  - — Ada delay
  - — Ada rendezvous
- Y A M—Yet Another Methodology unless
  - — laboratory work is integral to method's approach
  - — pedigree of examples is laboratory proven

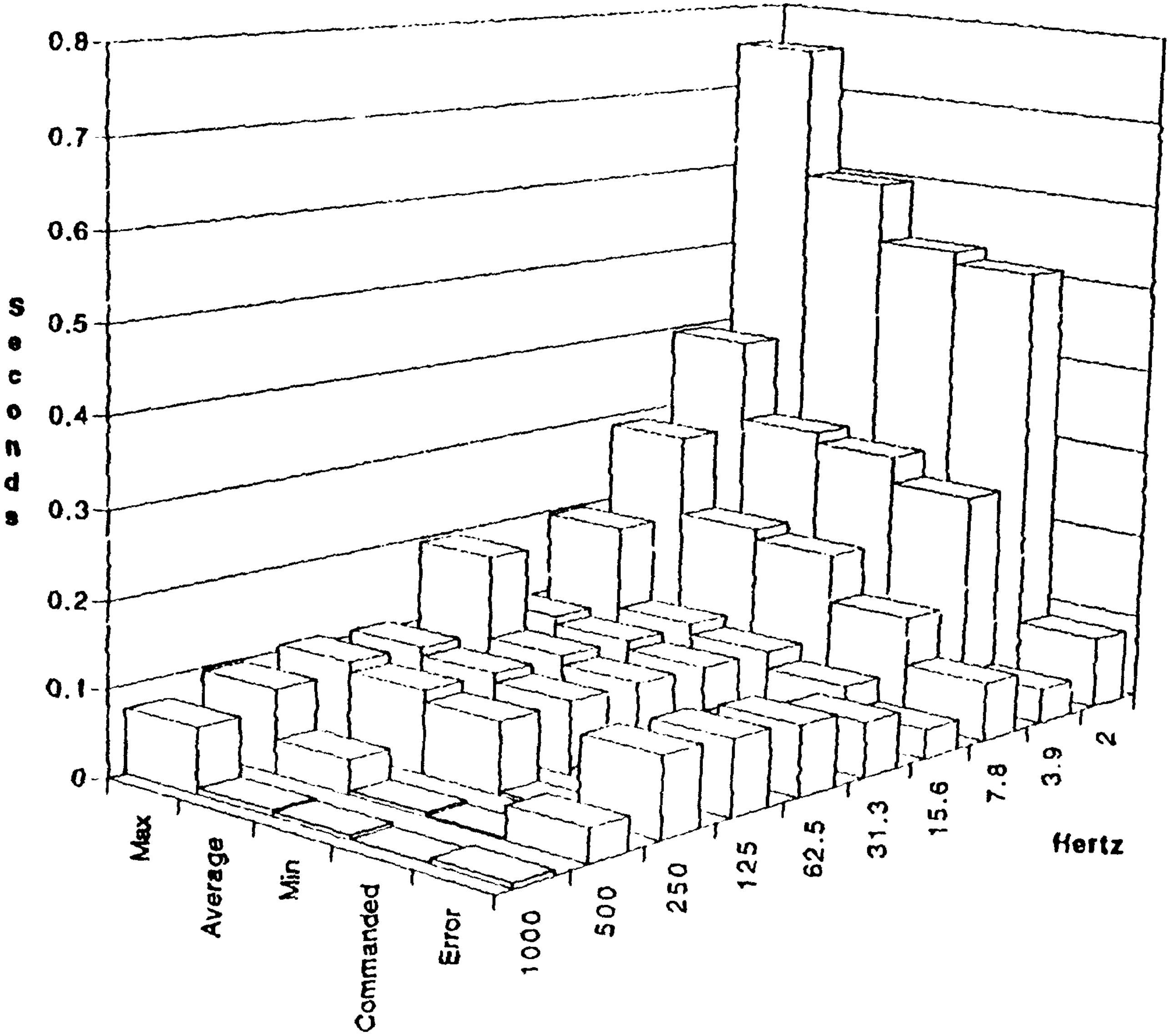
Alsys / Lynx OS on PS/2 Ada Delay



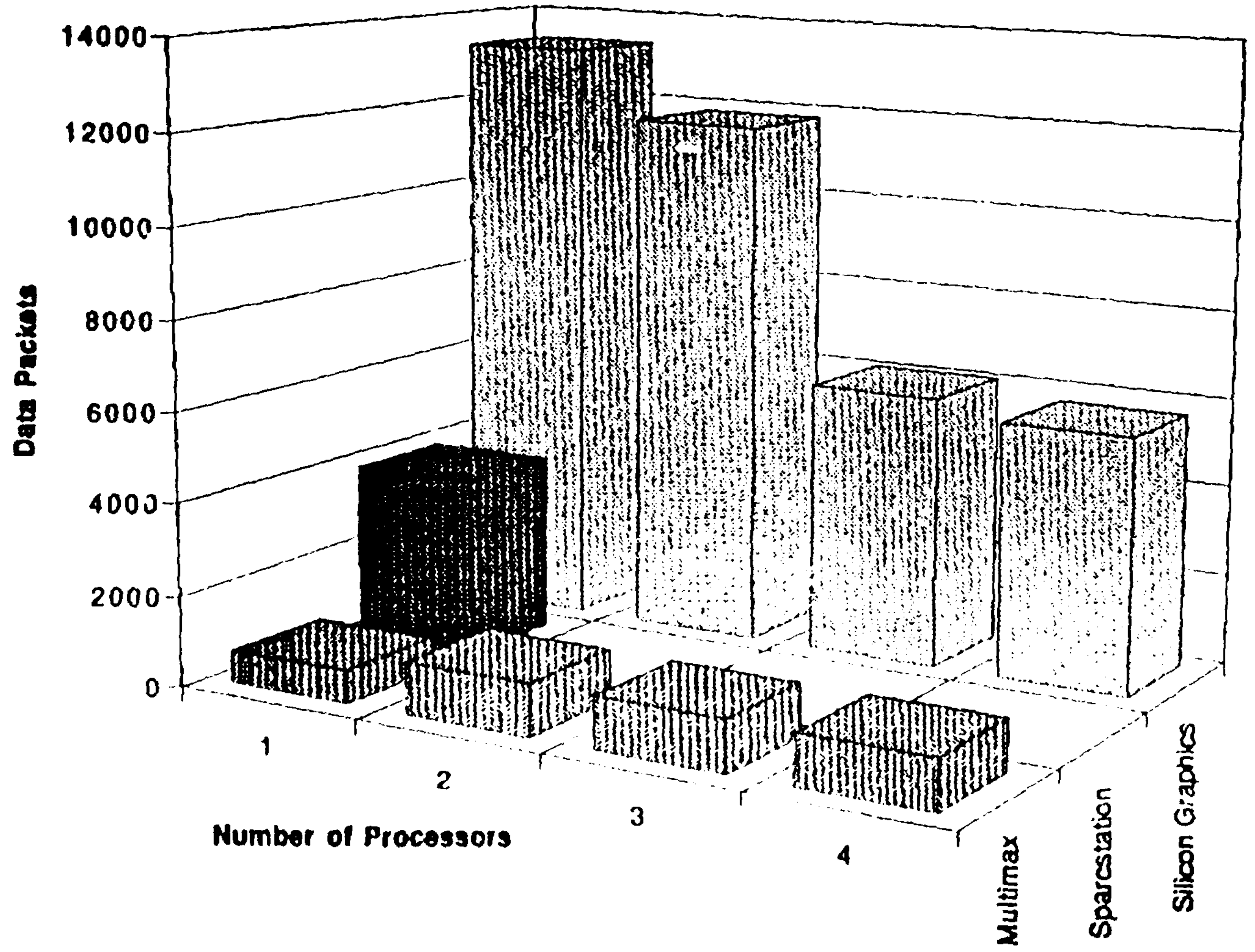
Telesoft / Unix - Sun Sparcstation 1



Verdix / Mach - Multimax



RTS3: A 20 Second Simulation



LaDPART Assessment 1

- Academic compartmentization of "real-time" definition
  - — subject defined by its problem perpective, i.e. deadlines and their treatment, rather than system response

- Real-time includes any program whose correctness (desirable behavior) depends on timeliness of its components

- Rate of response and stability of this response under operational load is an alternative real-time design driver

LaDPART Assessment 2

- LaDPART Systemic Issue of Computer Technology overshadows
  - —Overall Design Approach
  - —Methodology
  - —Scheduling Theory

- Recommendation
  - — Identify likely candidates and their relative contribution to the problem
  - — Develop a test case scenerio
  - —Sponsor a competition of approaches

Real-Time Design Methods in the current literature

- Family of extensions to structured analysis and design such as SDRTS (Ward & Mellor, 1985), DARTS (Gomaa, 1986), Cohesion and Coupling extensions (Nielsen & Shumate, 1988), etc.

- Family of extensions to object-oriented design such as OOD as popularized by Booch (1983), temporal objects as presented by Agrawala (1990), Case World R-T Conf. (Boston 1992) had 6 speakers on OOD for real-time systems.

- Family of process modeling methods such as (Hufnagel & Brown, 1989), (Howes & Weaver 1989), (Howes, 1990), and (Sanden, 1990).

Current Real-Time design theories

- Give little information on the timing behavior of the real-time system being designed.
- Rely on independent analysis of the design, or testing of an implementation (often a simulation or prototype) to gain an understanding of the system's timing behavior.

Current real-time scheduling theories

- Are based on the time-line model.
- Assume the system is already designed.
  - execution times of all tasks are known
  - dead-lines for all tasks are known
  - most tasks are periodic with occasional asynchronous request
- Restrict most analysis to periodic tasks.
- Introduce significant overhead to insure predictable timing behavior.
- Scheduling theorists often recommend "time budgeting" to "design in" predictable timing behavior.

The time-line model

- The model used in the study of scheduling theory.
- The model on which real-time systems with cyclic executives are based.
- The model on which the majority of deployed defense real-time systems is based.

Strengths of the time-line model

- Useful for reasoning about whether dead-lines of certain task sets will be met.
- Useful for reasoning about which dead-lines will not be met under certain conditions.

Disadvantages of the time-line model

- Most real-time practitioners have identified this model with the real-world, real-time problem space.
- It models the time frame in which a real-time problem is solved rather than the problem itself.
- It is not a good abstraction for reasoning about which functions should reside in which tasks.
- It is not a good abstraction for reasoning about how concurrent tasks should be designed.
- It does not generalize well to multiprocessors.

Evolution of real-time models

- Function Driven Scheduling
  - Time Value Functions (Jensen, et. al., 85 - 93)
  - Importance Abstraction (Strayer, 92) -- no deadlines
- Generalizations of Structured Design
  - SDRTS (Ward & Mellor, 85)
  - DARTS (Gomaa, 86)
  - Cohesion & Coupling extensions (Nielsen & Shumate, 1988)
- Generalizations of Object-Oriented Design
  - Temporal objects (Agrawala, 90)
- Process Modeling
  - Physical concurrency (Howes & Weaver, 1989), (Howes, 90)
  - Vertically Partitioned Objects (Hufnagel & Brown, 1989)
  - Entity Life Modeling (Sanden, 1990)

Quotation: Stephen Hufnagel & James Brown, IEEE TOSE, 15(1989) 8, p. 935

"It has long been believed that object-oriented system structuring offers advantages for design and implementation of computer software systems with respect to comprehensibility, verifiability, maintainability, etc.

The major obstacle to widespread use of object-oriented systems has been that their execution has been intrinsically inefficient due to excessive overhead."

Quotation: Stephen Hufnagel & James Brown, IEEE TOSE, 15(1989) 8, p. 938

"Object-oriented systems traditionally result in a workload that contains a great deal of context switching and data movement, thereby reducing performance.

This increased workload arises because of the smaller granularity of system structures resulting from object-oriented design.

Much data movement is among related object data structures."

Claims for real-time design methodologies by their authors

- Maintainability
- Reliability
- Transportability
- Resuse

Do these methodologies live up to their claims? -- our findings

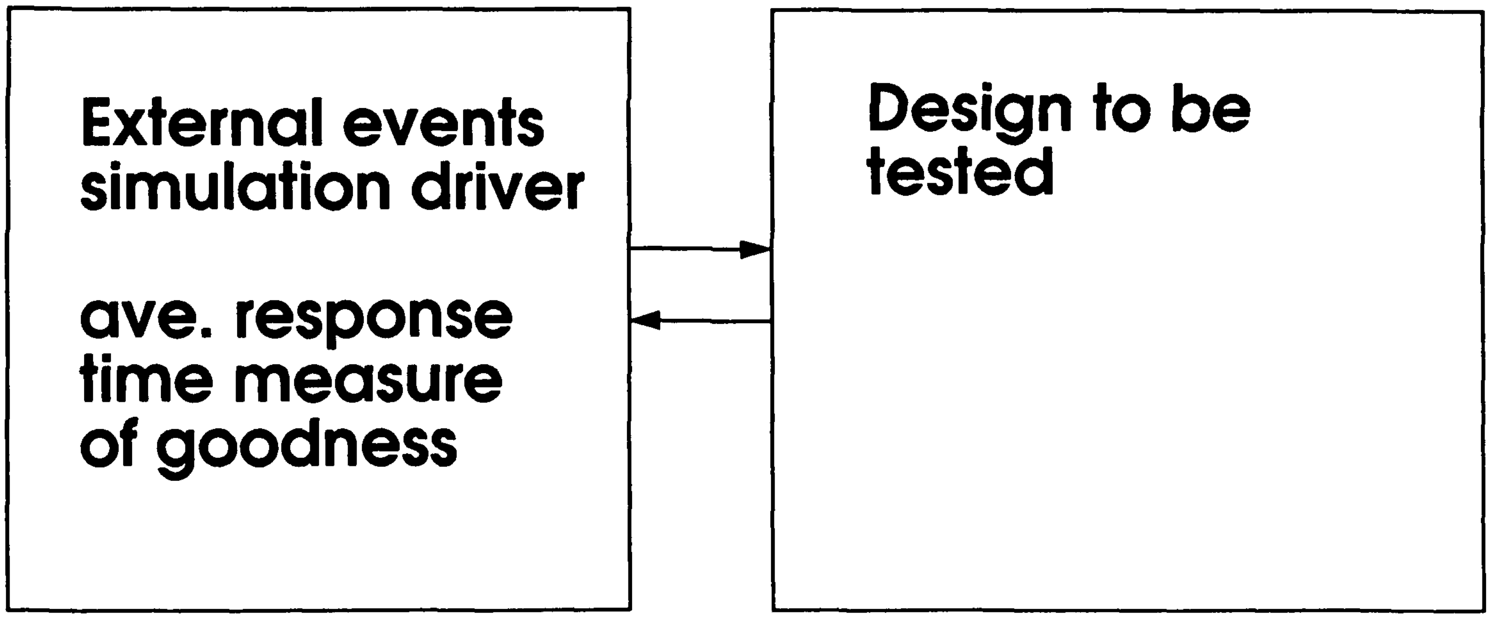
- They do not explicitly address performance or timing behavior.
- Most are not based on experimentation.
- Some have a tendency to lead to complex designs that can be shown to be inefficient.
- Sime do not deliver on their claims with regard to maintainability and transportability. We are not sure about reuse but we suspect the answer is the same.
- Despite these shortcomings they appear to offer a more natural approach to better designs than approaches like time budgeting or RMA.

Some experiments we conducted

- Redesign of Nielsen & Shumate's RTS example using the process modeling technique and comparison with their design on a sequential machine (inability to make a fair comparison on our parallel machine).
- Redesign of Hassan Gomaa's Robot controller example using the process modeling technique and comparison with his design on a sequential machine.

How the comparisons were made.

- The same (external events) simulation driver is used to test all of the designs to be compared. Thus, all designs must have the same interface to the simulator.

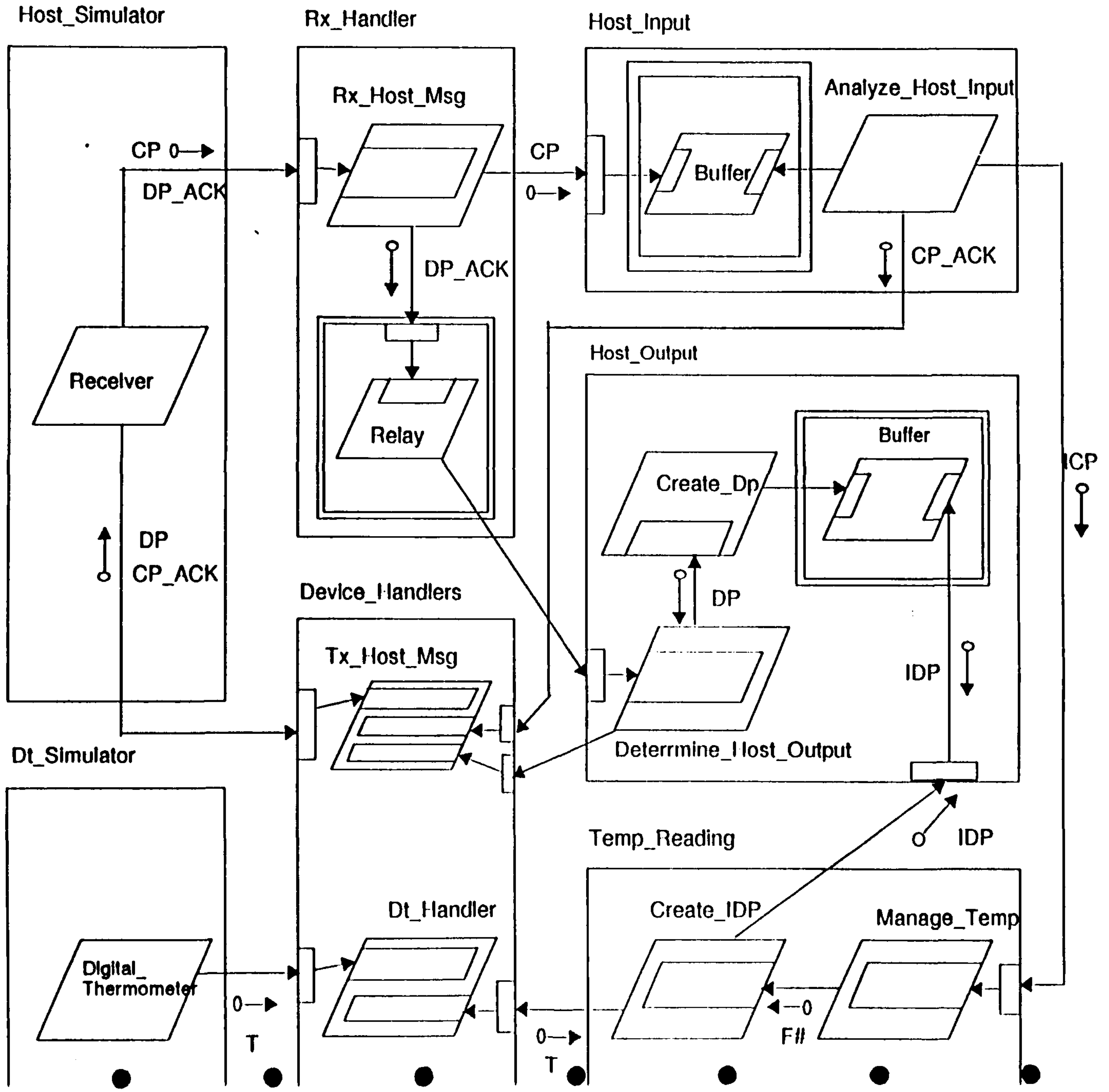


External events simulation driver

ave. response time measure of goodness

Design to be tested

ORGINAL NIELSEN & SHUMATE DESIGN OF THE REMOTE TEMPERATURE SENSOR



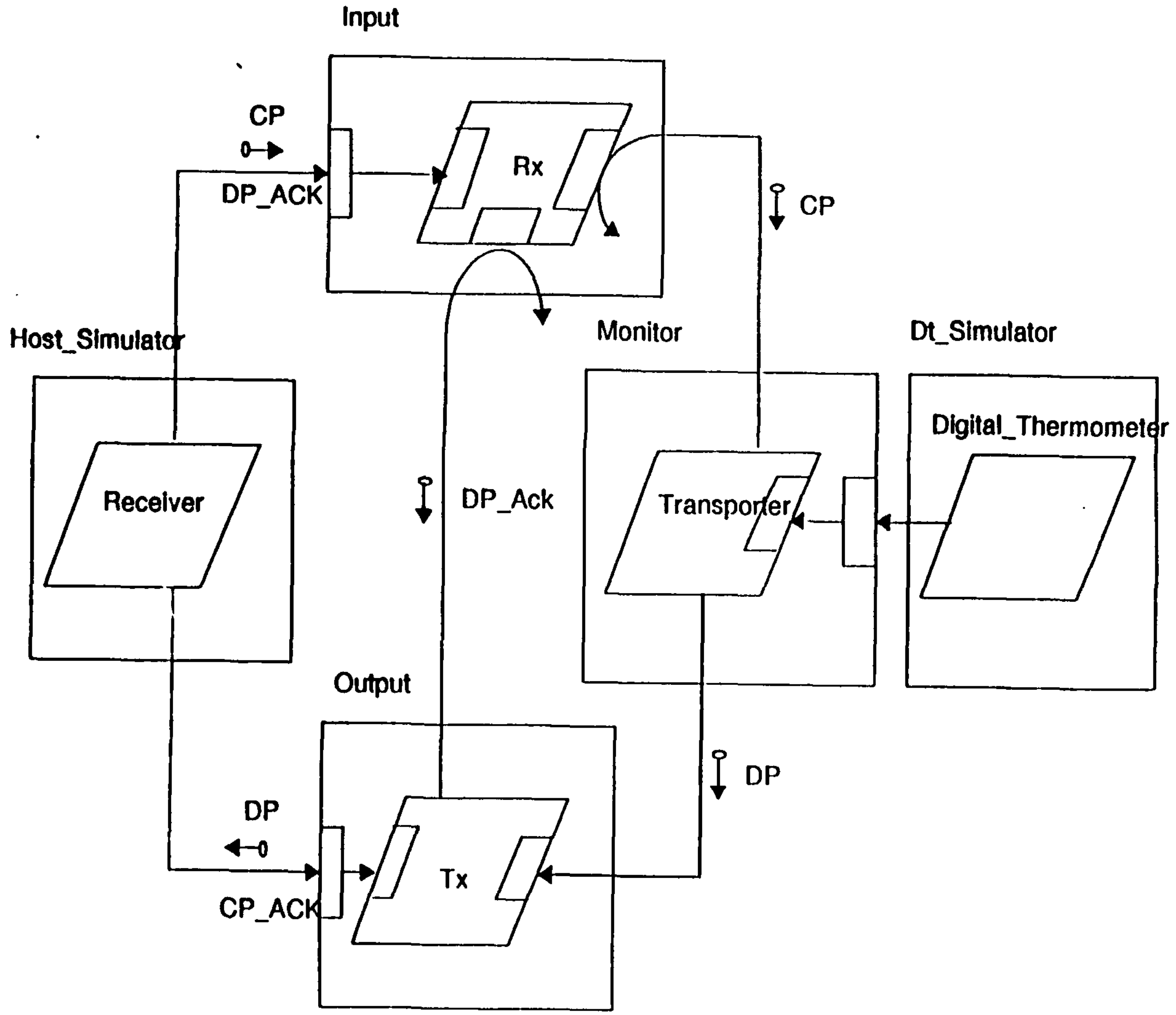
The process modeling design principles

- Physical Concurrency (structuring principle)
- Reducing mean service time of cyclic functions (tuning principle)

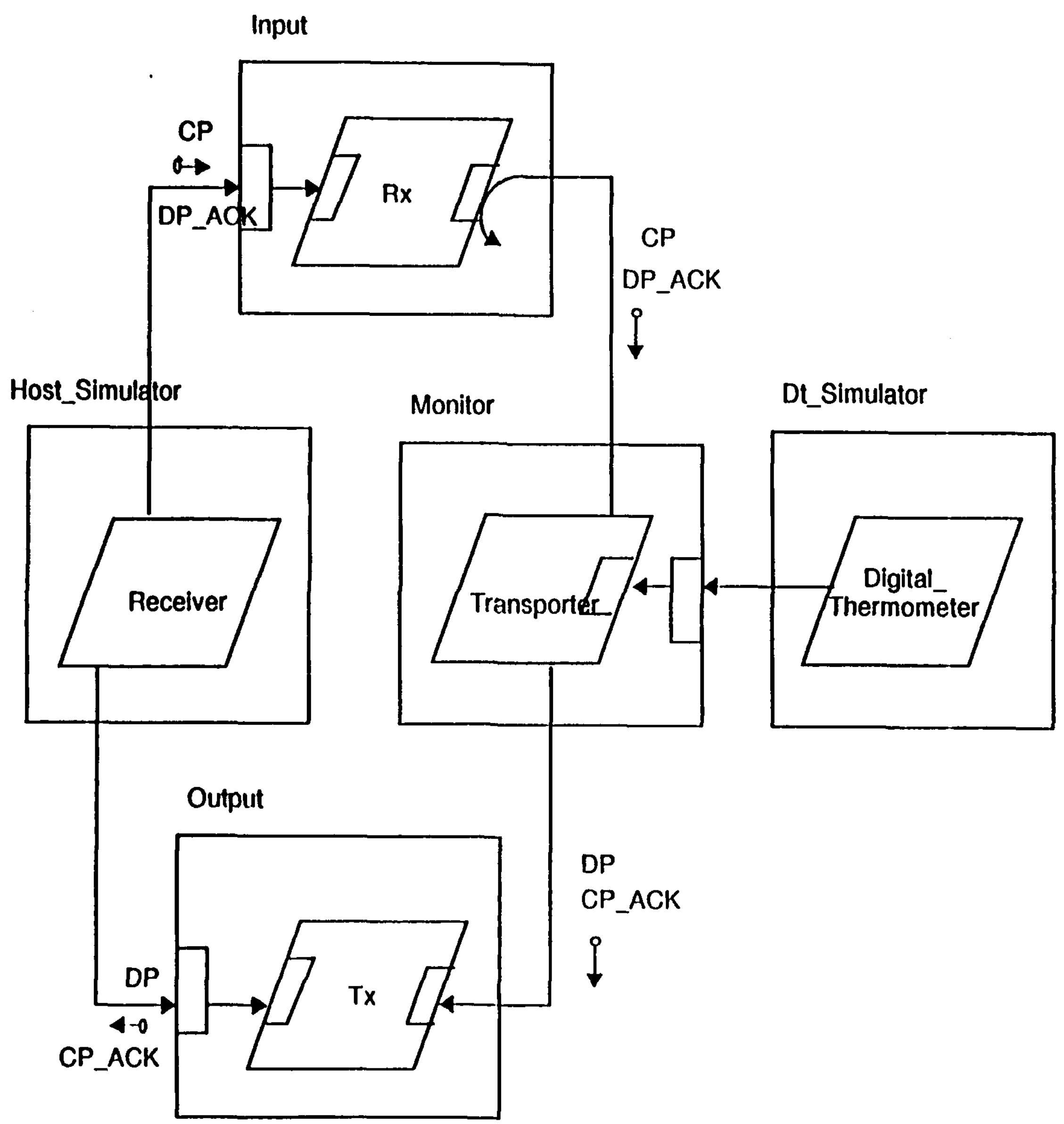
Ways of introducing concurrency into a design

- Maximal concurrency
  - primary goal for classical supercomputer applications
  - to reduce total computation time without regard for efficient processor utilization
- Conceptual concurrency
  - to support design abstractions
  - principle type of concurrency found in most real-time design methods
- Physical concurrency
  - to reduce complexity while still retaining as much concurrency as exists naturally in the problem space
- Minimal concurrency
  - to optimize the performance on sequential machines and to minimize context switching

FIRST REDESIGN OF THE REMOTE TEMPERATURE SENSOR SYSTEM



DESIGN ITERATION BASED ON REDUCING MEAN SERVICE TIMES



Results of the experiment on SUN 3/50 made in 1989

- Original N & S -- 8 readings/sec.
- Tuned N & S -- 30 readings/sec.
- First 3 task solution -- 73 readings/sec.
- Second 3 task solution -- 82 readings/sec.
- 3 tasks with blocking int. handler -- 110 readings/sec.
- 1 task solutions -- 120 - 180 readings/sec.

Results of experiments on Encore 320 during 1989 - 1993

- In 1989 the 3 task model ported to the Encore simply by recompiling, but the N & S version could not be made to run without deadlocking. Once, a 20 second comparison was made and the 3 task version was significantly faster.

- In 1993 both versions ported to the Encore simply by recompiling.

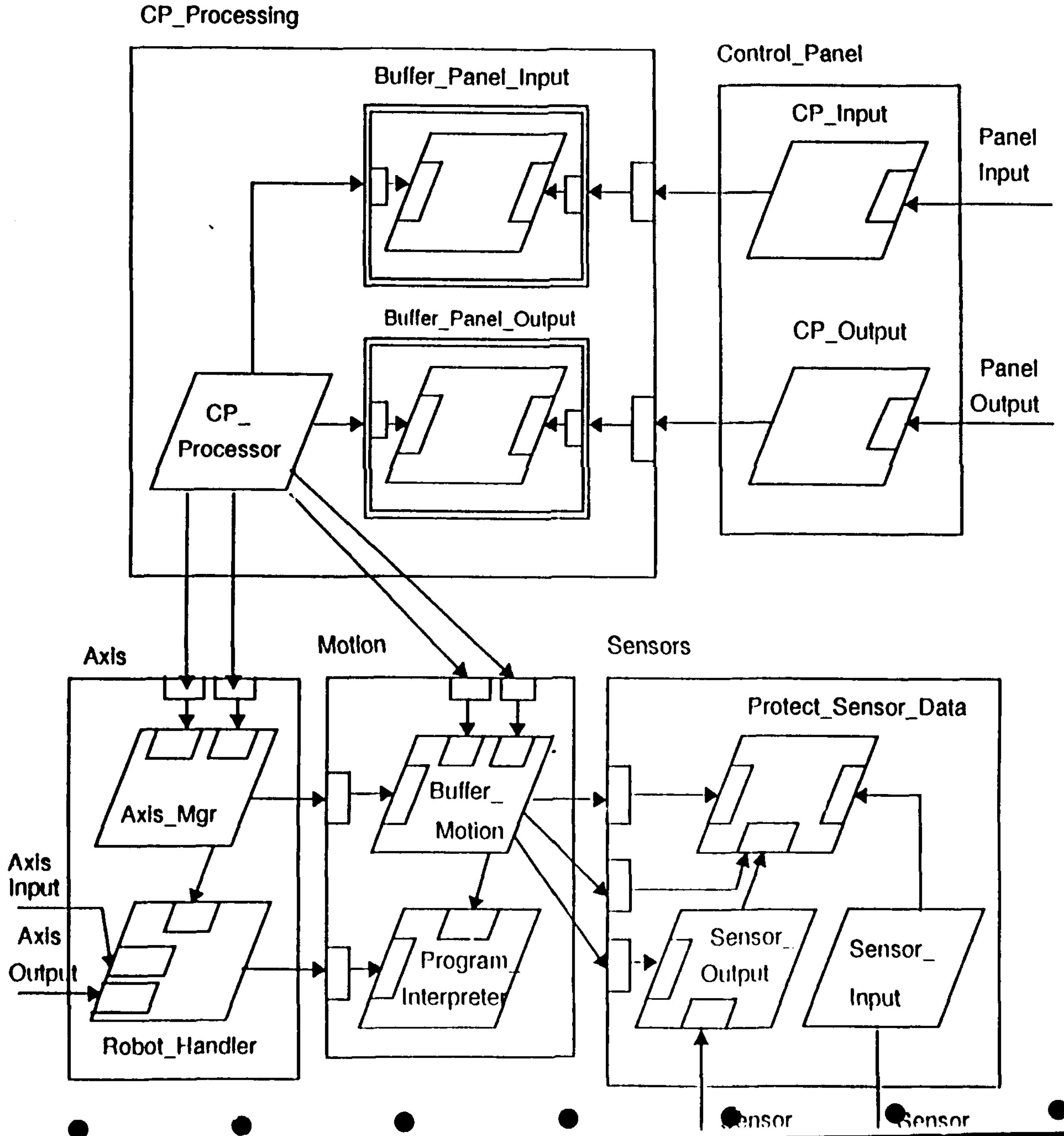
- Preliminary runs indicate that the results are similar to the results on the SUN 3/50. Despite the number of processors (14), no speed up is observed after 3 processors and the cumulative performance is not better than that of a single processor SUN 3/50.

- We are currently porting the simulation to an Encore 9300 with a real-time operating system.

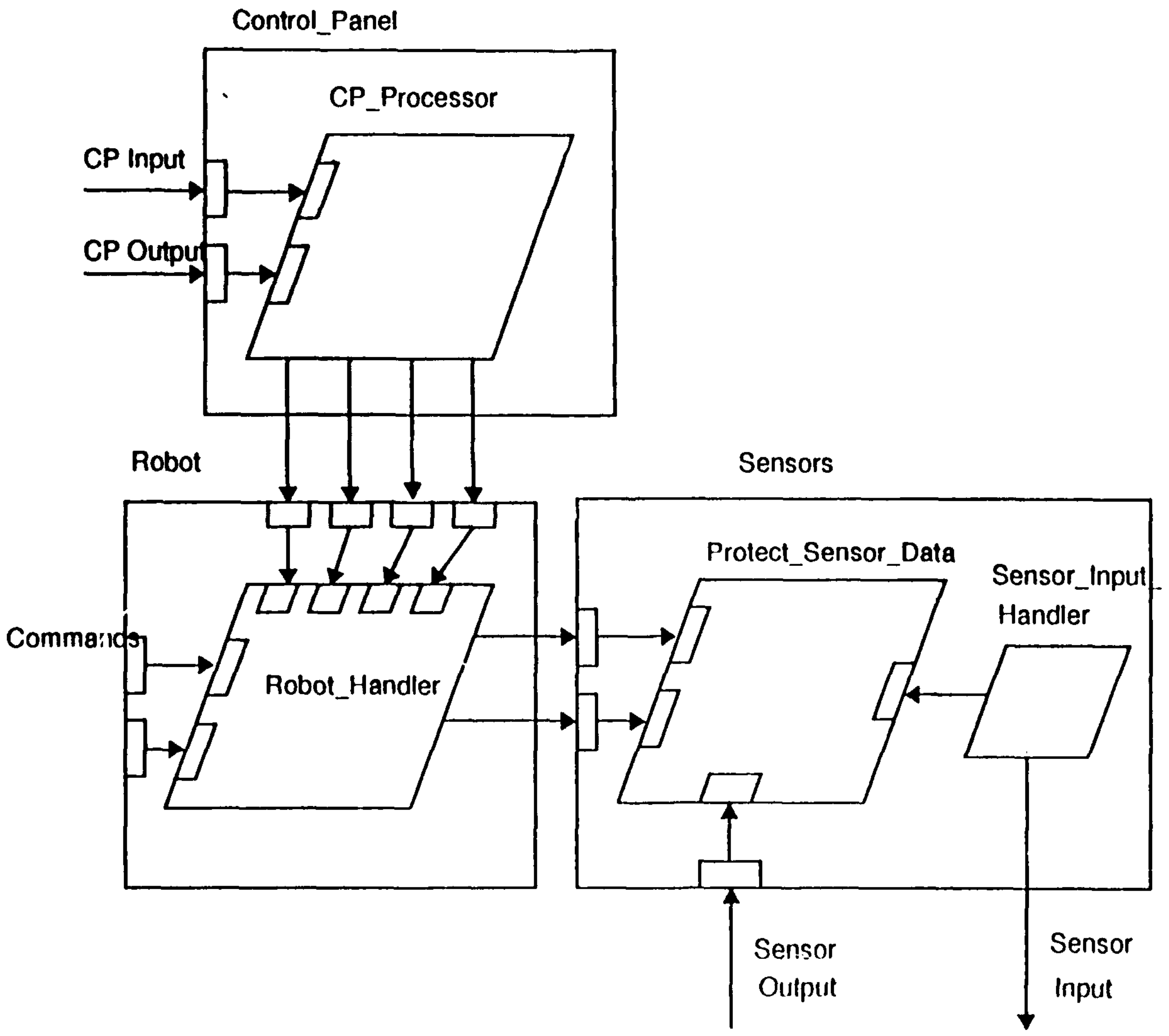
DARTS approach

- Based on data flow diagrams and heuristics for combining transforms on the diagrams into tasks.
- Data flow diagrams do not contain timing information.
- Heuristics do not emphasize timing.
- Heuristics are not consistent.
- Heuristics lead to a large number of tasks.

N & S/GOMAA'S DESIGN OF THE ROBOT CONTROLLER SYSTEM



REDESIGN OF THE ROBOT CONTROLLER SYSTEM USING NEW PRINCIPLES



Results of the experiment on a VAX 8800 in 1989

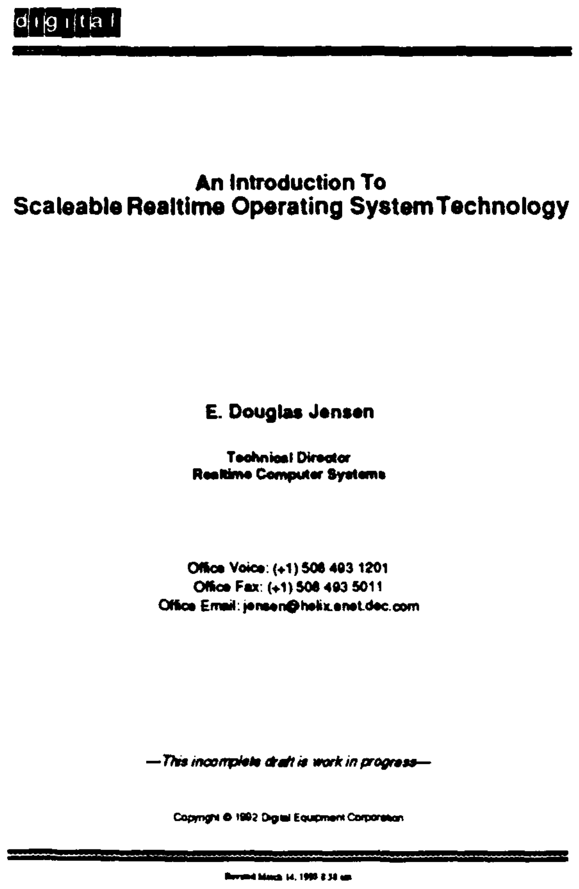
- Gomaa Design
  - 100% MOTION COMMANDS -- 794
  - 75/25% MIXED COMMANDS -- 886
  - 50/50% MIXED COMMANDS -- 900
  - 100% SENSOR COMMANDS --- 1,341
- Student Design
  - 100% MOTION COMMANDS -- 2,179
  - 75/25% MIXED COMMANDS -- 2,179
  - 50/50% MIXED COMMANDS -- 2,162
  - 100% SENSOR COMMANDS --- 2,242

Other observations about these experiments

- N & S and Gomaa designs of the RTS and Robot controller systems had 4 to 6 times more source code than any of the alternate designs we came up with.
- Recompilation times for the N & S and Gomaa designs were always at least 3 times longer.
- Porting the 3 task version to several other architectures was considerably simpler than porting the N & S version.

Conclusions

- Most real-time design methodologies have not been tested sufficiently to determine their value.
- Many real-time methods are advanced without experimental confirmation.
- The design principles of physical concurrency and minimizing mean service time of cyclic functions seems to work better than the methods they were tested against.
- These two principles do not suffice to decide all design questions for a real-time system.
- The physical concurrency principle has a conceptual parallel with object oriented design.



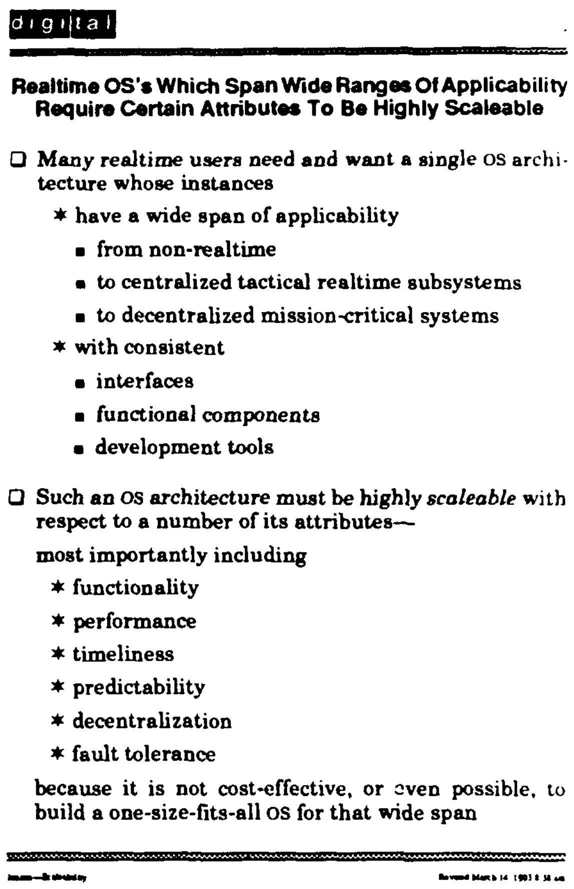
An Introduction To Scaleable Realtime Operating System Technology

E. Douglas Jensen

Technical Director
Realtime Computer Systems

Office Voice: (+1) 508 493 1201
Office Fax: (+1) 508 493 5011
Office Email: jensen@helix.enet.dec.com

—This incomplete draft is work in progress—



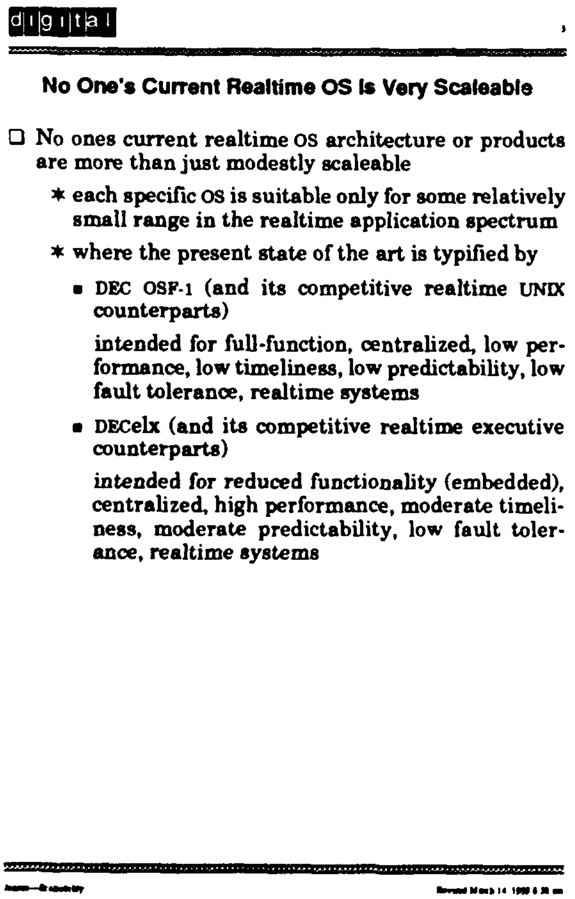
Realtime OS's Which Span Wide Ranges Of Applicability Require Certain Attributes To Be Highly Scaleable

❑ Many realtime users need and want a single OS architecture whose instances
- ∗ have a wide span of applicability
  - from non-realtime
  - to centralized tactical realtime subsystems
  - to decentralized mission-critical systems
- ∗ with consistent
  - interfaces
  - functional components
  - development tools

❑ Such an OS architecture must be highly *scaleable* with respect to a number of its attributes—

most importantly including
- ∗ functionality
- ∗ performance
- ∗ timeliness
- ∗ predictability
- ∗ decentralization
- ∗ fault tolerance

because it is not cost-effective, or even possible, to build a one-size-fits-all OS for that wide span

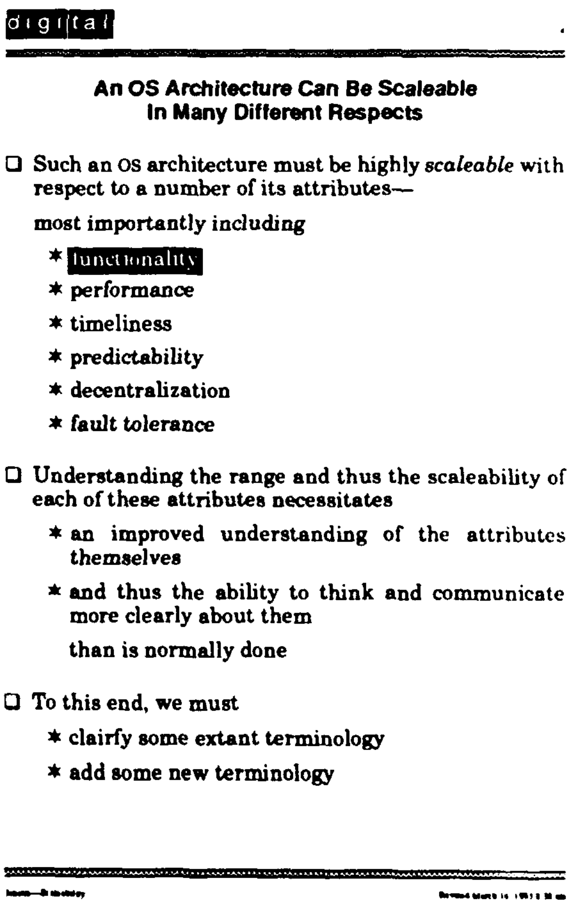


No One's Current Realtime OS Is Very Scaleable

❑ No ones current realtime OS architecture or products are more than just modestly scaleable
- ∗ each specific OS is suitable only for some relatively small range in the realtime application spectrum
- ∗ where the present state of the art is typified by
  - DEC OSF-1 (and its competitive realtime UNIX counterparts)

    intended for full-function, centralized, low performance, low timeliness, low predictability, low fault tolerance, realtime systems
  - DECelx (and its competitive realtime executive counterparts)

    intended for reduced functionality (embedded), centralized, high performance, moderate timeliness, moderate predictability, low fault tolerance, realtime systems



An OS Architecture Can Be Scaleable In Many Different Respects

❑ Such an OS architecture must be highly *scaleable* with respect to a number of its attributes—

most importantly including
- ∗ functionality
- ∗ performance
- ∗ timeliness
- ∗ predictability
- ∗ decentralization
- ∗ fault tolerance

❑ Understanding the range and thus the scaleability of each of these attributes necessitates
- ∗ an improved understanding of the attributes themselves
- ∗ and thus the ability to think and communicate more clearly about them

  than is normally done

❑ To this end, we must
- ∗ clairfy some extant terminology
- ∗ add some new terminology

OS Functionality Is Scaleable To The Extent That It Can Be Changed Coherently Over The Spectrum

❑ OS—particularly realtime OS—functionality ranges from

- ∗ none—some realtime systems have no OS at all
- ∗ to simple embedded realtime subsystems with minimal OS (executive) functionality
- ∗ to complex realtime systems with extensive OS functionality

❑ An OS architecture is functionally scaleable to the extent that its instantiations'

- ∗ functionality can be increased and decreased—e.g., to match the requirements of
  - ▪ the applications
  - ▪ trade-offs for other attributes
- ∗ functional interfaces increase and decrease coherently (i.e., superset and subset) with the functionality
- ∗ code base increases and decreases, instead of having to be replaced, with the functionality

❑ Functional scaleability may be static (at configuration time) or dynamic (at execution time)

❑ Functional scaleability is particularly important because scaleability of other OS attributes depends on it

An Example Of Highly Scaleable OS Functionality Is The Use Of A Microkernel-Based OS

❑ A microkernel can be the basis of an OS architecture with high functional scaleability

❑ A microkernel can support (normally statically)

- ∗ different functional "servers" as needed
  - ▪ OS—e.g., diskless executive, reduced functionality OS, full UNIX
  - ▪ system software—e.g., networking, data management
  - ▪ application software
- ∗ different API's ("personalities")—even at the same time
- ∗ direct use of the native kernel interface

❑ A microkernel (like any kernel) can be designed to permit system or application software which has been developed in user space

to be executed in the kernel address space

- ∗ this improves performance of time-critical code
- ∗ without the higher costs of developing kernel code

(although this is heretical to some microkernel purists—at least outside of the realtime field)

An Example Of Highly Scaleable OS Functionality Is The Use Of Policy/Mechanism Separation

❑ The separation of resource management

- ∗ mechanisms
  - ▪ standard application-neutral building blocks
  - ▪ e.g., in the kernel
- ∗ policies
  - ▪ application-specific algorithms
  - ▪ e.g., in clients of the kernel

is an effective technique for achieving a high degree of functional scaleability

❑ For example, policies can be separated from mechanisms for

- ∗ processor scheduling
- ∗ transactional data management—correctness, concurrency control, permanence
- ∗ secondary storage—consistency, recovery, interface semantics (e.g., objects, files)
- ∗ virtual memory—paging

❑ Policies may be selected dynamically (i.e., at execution time) or statically (i.e., prior to execution time)

❑ This is essentially the dual of the more widely used principle of information hiding

An OS Architecture Can Be Scaleable In Many Different Respects

❑ Such an OS architecture must be highly *scaleable* with respect to a number of its attributes—

most importantly including

- ∗ functionality
- ∗ performance
- ∗ timeliness
- ∗ predictability
- ∗ decentralization
- ∗ fault tolerance

Realtime OS Performance Is Commonly Measured In Terms Of The Magnitudes Of Response Times

❑ Commercial realtime OS suppliers and users commonly consider *performance* in terms of the magnitude of the time to initiate a computation which is

- ∗ newly released—the metric is "interrupt response time"

  (preemptive scheduling is the norm)

- ∗ already released—the metric is the "context switch time" subset of interrupt response time

  (regardless of whether scheduling is

  - ▪ preemptive—the usual case
  - ▪ or non-preemptive)

❑ Performance as interrupt response time presumes the fact that most realtime OS's are *globally asynchronous* in the sense of being event-driven;

a minority of OS's seek determinism by being *globally synchronous* in the sense of being time-driven,

and thus don't have interrupts

By These Metrics, High Performance Is 10-100 μSec

❑ By these metrics, using contemporary processor speeds

- ∗ "high" realtime performance is typically regarded to be on the order of
  - ▪ 10 microseconds for limited-functionality executives
  - ▪ 100 microseconds for full-functionality OS's
- ∗ "low" realtime performance is typically regarded to be on the order of 1000 microseconds and more

Realtime OS Performance Is Best Measured In Terms Of The Magnitudes Of Computation Completion Times

❑ The performance of realtime OS's is more usefully measured in terms of

- ∗ the end
- ∗ rather than one of the means:

the magnitude of the computation <u>completion</u> times (such as deadlines)

which can be attained with given hardware

❑ This is the performance metric that deterministic OS's use—

and globally synchronous ones must use (since they have no interrupts)

A Realtime Computation Has A Time Constraint

❑ We define a *realtime computation* to be a segment of a computational entity (such a thread, task, or process) subject to a time constraint

❑ A *time constraint* is the relationship between

- ∗ when a realtime computation completes execution
- ∗ the *temporal merit* of that computation

e.g., in the classical deadline case

- ∗ completing before the deadline time is better
- ∗ completing after the deadline time is worse

❑ A time constraint is manifest in the computation program as a demarcated region of code whose execution completion time is subject to the time constraint—

e.g., the computation must complete execution of the region before the deadline time arrives

```
BEGIN TC (DL = 30 mS)
   .......  ⎫
   .......  ⎪
   .......  ⎪
   .......  ⎬ Realtime Computation
   .......  ⎪
   .......  ⎭
END TC
```

otherwise it must suffer an exception condition

The Magnitudes Of Completion Time Constraints Are Not Specified For The Majority Of OS's

❑ The performance of realtime OS's in terms of the magnitude of the computation completion time constraints is usually unspecified

because computation completion time constraints are

- ∗ not yet supported directly by most commercial OS products, which deal instead with
  - ▪ starting computations as fast as possible
  - ▪ "priorities"—whose semantics are user-defined
- ∗ left to the users to satisfy by assigning and manipulating priorities and resources

❑ This is due to

- ∗ the historical context of realtime computing
  - ▪ the paucity of processing power and memory—
    made direct control by the user necessary
  - ▪ the simplicity of low-level sampled-data centralized subsystems—
    made direct control by the user possible
- ∗ user and vendor habituation despite evolutionary changes in both aspects of that context

❑ However, computation completion time constraints in the form of deadlines—e.g., priority ceiling protocols for rate-monotonic schedulinp—have now appeared in POSIX 1003.4B

Higher Performance Of Means Is Usually Necessary But Not Sufficient For That Of Ends

❑ Higher performance in the sense of interrupt response time is

- ∗ necessary (in the usual asynchronous cases)
- ∗ but not sufficient

for higher performance in the sense of computation completion times

❑ An effective technique for improving performance in the sense of interrupt response time is kernel preemptability

❑ A key factor in performance in the sense of computation completion times

is how resource dependencies and conflicts among computations are resolved—e.g., by the scheduler

OS Performance Is Scaleable To The Extent That It Can Be Changed At Lower Functional Granularity

❑ A OS architecture's performance is scaleable to the extent that its instantiations' performance

- ∗ can be increased and decreased in magnitude—e.g., to match the requirements of
  - ▪ the applications
  - ▪ trade-offs for other OS attributes
- ∗ at lower levels of functional granularity—e.g., of the
  - ▪ individual OS or application computations
  - ▪ vs. OS functions or services
  - ▪ vs. the OS as a whole

Functional ScaleabilityAffects PerformanceScaleability

❑ Some kinds and degrees of performance scaleability are more advantageous

if they are achieved without affecting functional interfaces

❑ Performance scaleability is generally facilitated by greater functional scaleability—e g.,

- ∗ higher realtime performance (in any sense) is easier to achieve in OS's having less
  - ▪ functionality in general
  - ▪ of certain functionality in particular
- ∗ realtime performance is generally greatly affected by resource management policies—e.g.,
  - ▪ scheduling
  - ▪ concurrency control and synchronization
  - ▪ virtual and physical storage management

  in any given application context

❑ Performance may be scaled

- ∗ statically—e.g., configuring a different file system
- ∗ dynamically—e.g.,
  - ▪ turning preemption on/off
  - ▪ selecting a different scheduling algorithm

Higher Performance Does Not Necessarily Imply Meeting All Lower Performance Requirements

❑ Providing higher performance in the sense of shorter interrupt response time

- ∗ also meets lower performance requirements
- ∗ which implies performance scaleability means only the ability to scale up to higher performance

❑ But providing higher performance in the sense of shorter computation completion times does not necessarily imply meeting lower performance requirements

- ∗ e.g., a particular realtime scheduling algorithm may provide acceptable completion times
  - ▪ for a set of computations whose time constraint magnitudes are relatively
    - ◊ short
    - ◊ uniform

    (and thus constitute a a higher performance requirement)
  - ▪ but not for a set of computations whose time constraint magnitudes are widely varied from short to long

    (which constitutes a lower performance requirement)
- ∗ this demonstrates that performance scaleability in general means both up and down—

  which calls for scaleable scheduling policies

An OS Architecture Can Be Scaleable In Many Different Respects

❑ Such an OS architecture must be highly *scaleable* with respect to a number of its attributes—

most importantly including

- ∗ functionality
- ∗ performance
- ∗ timeliness
- ∗ predictability
- ∗ decentralization
- ∗ fault tolerance

Timeliness Is The Basis For Realtime Scheduling

❑ We consider the *timeliness*—i.e., *temporal merit*—of computations to be the principle basis for

- ∗ specifying
- ∗ scheduling
- ∗ evaluating

computation completion times

❑ We define timeliness with a framework consisting of three relationships (e.g., functions)

A Timeliness Framework Is Comprised Of Three Parts

❑ Each realtime computation has a *time constraint*—i.e., a relationship between

- ∗ when the computation completes execution
- ∗ the resulting temporal merit—timeliness—of that computation

(e.g., for the classsical deadline time constraint, *lateness* = completion time − deadline)

❑ A *collective temporal merit* relationship defines

- ∗ the collective timeliness of a set of computations
- ∗ in terms of the individual timeliness of all its constituent computations

(e.g., the number of deadlines met—i.e., with negative lateness)

❑ A *collective temporal acceptability* relationship defines

- ∗ the *acceptability*—in an application-specific metric
- ∗ of the completion times—predicted or experienced—for a set of computations

  expressed in terms of their individual or collective timeliness
- ∗ for specified system and application states

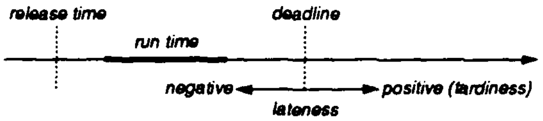
(e.g., acceptable means always meeting all deadlines)

Timeliness For Classical Deadline Time Constraints Is In Terms Of Tardiness

❑ The classical deadline time constraint (i.e., in scheduling theory) employs

  * *lateness* = completion time − deadline
  * or *tardiness* = positive lateness

as its individual measure of timeliness



❑ The collective timeliness relationship of a set of computations having classical deadline time constraints is most frequently chosen to be one of the following

  * the occurrence or not of at least one tardy (positive lateness) completion
  * the number of tardy completions
  * the mean lateness

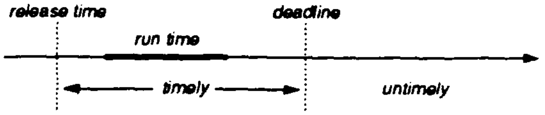
❑ Classical deadline-based scheduling theory often implicitly presumes that

collective temporal acceptability is equivalent to collective timeliness

The Traditional Hard Deadline Case Allows Only For Binary Timeliness And Acceptability

❑ The traditional realtime computing interpretation of "hard" deadlines implies restrictions of timeliness to

  * a binary special case of the deadline time constraint—timely and untimely



  * a binary collective timeliness relationship
    - untimely: the occurrence of at least one tardy completion
    - timely: otherwise
  * a binary measure of collective temporal acceptability
    - acceptable: no occurrence of tardy completions (unanimous optimum) under any conditions
    - unacceptable: the occurrence of at least one tardy completion under any conditions

  where the semantics of "unacceptable" are specific to the computation and application—e.g.,

    - non-productive
    - counter-productive

  in some way

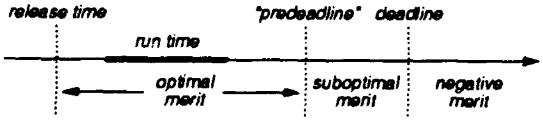
Often Time Constraints Are Not Binary

❑ Often it is very useful or necessary to have *softer*—i.e., non-binary—time constraints

❑ A common example of such a softer time constraint:

if a particular computation cannot be completed at a time of optimal merit—i.e., before its "predeadline"

  * completing it a little "tardy" has reduced merit—but is better than not completing it at all
  * however, completing it actually tardy (after its deadline) has negative merit—i.e., is worse than not completing it at all



❑ Some softer time constraints are routinely handled in terms of lateness with scheduling theory—

but the linearity of lateness greatly limits the interpretation of merit (e.g., excludes this example)

❑ Realtime computing practice tends to express and handle softer time constraints even less effectively—

not on a time con·traint basis at all, but instead in disparate, ad hoc, imprecise ways

Often Collective Timeliness Is Not Binary

❑ Softer time constraints necessitate correspondingly "softer"—i.e., non-binary—collective timeliness relationships

❑ Using the previous time constraint example,

the collective timeliness relationship could be one which (as a scheduling criterion) increases the number of completions in the optimal region—e.g.,

  * the sum (or mean) of
  * weighted lateness = (completion time − deadline) + k (completion time − predeadline)

❑ Some softer collective timeliness relationships are routinely handled in terms of lateness with classical scheduling theory

while others necessitate more expressive time constraint relationships

❑ Realtime computing practice tends to express and handle softer collective timeliness less effectively—

not on a time constraint basis at all, but instead in disparate, ad hoc, imprecise ways

Often Temporal Acceptability Is Not Binary

❑ Softer collective timeliness necessitates correspondingly "softer"—i.e., non-binary—temporal acceptability relationships

❑ The degree of collective temporal acceptability might be based on

- ∗ collective timeliness alone—e.g., acceptable only
  - ▪ only above one lower bound under certain circumstances, and above a different lower bound under other circumstances
- ∗ both individual and collective timeliness—e.g., acceptable to the degree that
  - ▪ some
    - ◊ total number of
    - ◊ or specific individual

    computations
  - ▪ are late by a certain amount
  - ▪ under certain conditions

❑ Realtime computing practice tends to express and handle softer temporal acceptability less effectively—

not on a time constraint basis at all, but instead in disparate, ad hoc, imprecise ways

Timeliness Is Scaleable To The Extent That The Choice Of The Three Relationships Is Unrestricted

❑ The timeliness of an OS architecture is the degree to which it supports the timeliness of

- ∗ application
- ∗ its own

computations

❑ The timeliness of an OS architecture is scaleable to the extent that its instantiations can accommodate

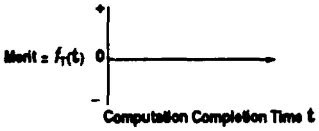
arbitrary (i.e., an unrestricted choice of) relationships for

- ∗ individual time constraints
- ∗ collective timeliness
- ∗ temporal acceptability

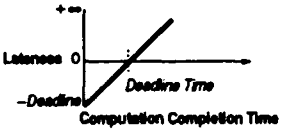
Scaleable Time Constraint Relationships Are Temporal Merit As A Function Of Completion Time

❑ The time constraint relationship can be made arbitrary by thinking explicitly of

individual temporal merit being any function f_T of the computation's completion time t



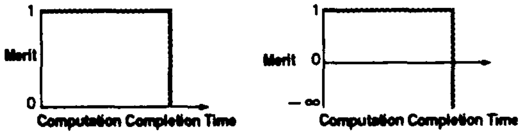
❑ The classical deadline function's merit of lateness is then depicted as



- ∗ a line
- ∗ with slope +1
- ∗ having a range of {–deadline, +∞}
- ∗ crossing the X axis at the deadline time (becoming tardiness)

The Traditional Realtime Computing Interpretation Of A Deadline Is A Downward Step Function

❑ The traditional realtime computing interpretation of a deadline, when viewed as a time constraint function, is



- ∗ a binary-valued, downward step function
  - ▪ completing the computation anytime between its release (X = 0) and deadline times is uniformly timely
  - ▪ and otherwise is uniformly untimely
- ∗ the smaller of the two binary merit values may be
  - ▪ 0: zero merit is attained for completing the computation after its deadline
  - ▪ –∞: a large merit penalty is incurred for completing the computation after its deadline

In Real Systems Very Often The Time Constraint Is Neither Linear Nor Binary

❑ Both the classical and traditional realtime computing interpretations of a deadline are often poor approximations to actual realtime constraints

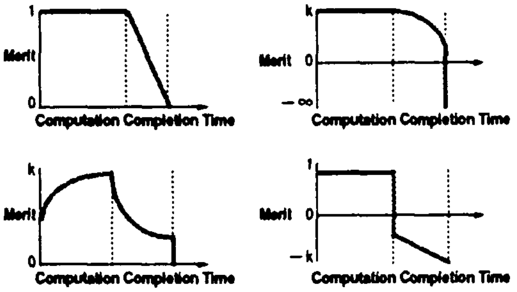
❑ There are many cases in realtime applications where

  ∗ there is some diminished merit attained for completing the computation within an allowable tardiness period

  ∗ the merit is not constant prior to the "deadline"

  ∗ the penalty is not constant after the "deadline"

  ∗ the merit measure and range are application-specific



❑ Deadlines are not a general mechanism for expressing scaleable realtime time constraints

Time Constraints Are Scaleable To The Extent That They Are Defined By Arbitrary Functions

❑ Time constraints are scaleable to the extent that they

  ∗ are arbitrary functions

  ∗ with arbitrary merit ranges

❑ The merit measure is application-specific and defined system-wide

❑ The computation completion time axis is the one the scheduler uses—it may be

  ∗ physical

    ▪ absolute ("calendar/wall clock") time

    ▪ relative to (since) some past event

  ∗ logical—a number which monotonically increases, but not necessarily at regular intervals

❑ The origin of the time constraint function axes is the current time (value of the system clock)

❑ Time constraint functions are

  ∗ derived by the programmers directly from the requirements and behavior of the realtime computation

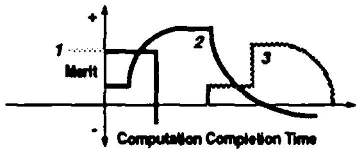
  (usually an application activity)

  ∗ subject to a system-wide engineering process

  (just as are assignments of classical priorities)

Collective Timeliness is The Scheduling Criterion

❑ A realtime scheduler considers all released time constraints between the current time and its horizon



❑ It assigns the estimated execution completion times—and consequently the

  ∗ initiation times

  ∗ partial ordering

for those computations

❑ It employs an algorithm chosen to optimize the collective timeliness—i.e., scheduling—criterion

(perhaps taking into account other factors such as dependencies)

❑ In general, collective timeliness is not necessarily unanimously optimum with respect to the individual computations' timeliness—

the traditional "hard" realtime computing criterion of all computations meeting their deadlines is a popular exception

Collective Timeliness Is Scaleable To The Extent That It Is Defined By Arbitrary Functions

❑ The collective timeliness relationship can be made arbitrary by thinking explicitly of

collective timeliness being any function f_c of the individuals' timeliness

❑ The common classical collective timeliness functions—e.g.,

  ∗ the number of tardy completions

  ∗ the mean lateness

can readily be generalized in terms of arbitrary individual merit measures

❑ The collective timeliness function f_c for traditional realtime computing's interpretation of hard deadline time constraints (assuming their usual range of $\{0,1\}$) is

f_c*: the product of the individual merits

Temporal Acceptability Is Scaleable To The Extent That It Is Defined By Arbitrary Functions

❑ The collective temporal acceptability relationship can be made arbitrary by thinking explicitly of

collective temporal acceptability being any function f_A of

- ✶ the individuals' and collective timeliness
- ✶ other parameters, such as system state

❑ The collective temporal acceptability function f_A for classical deadline time constraints is often null—i.e.,

$f_A(\ldots) = f_C(\ldots)$

❑ The collective temporal acceptability function f_A for traditional realtime computing's interpretation of hard deadline time constraints is

$f_A(\ldots) = f_C{}^*(\ldots)$

Highly Scaleable Timeliness Is Facilitated By Scheduler Policy/Mechanism Separation

❑ Highly scaleable timeliness is facilitated by a form of functional scaleability—scheduler policy/mechanism separation

because the scheduling policy

- ✶ employs the time constraint functions
- ✶ to optimize the collective timeliness function
- ✶ so as to meet the collective temporal acceptability function criterion

❑ The extent to which timeliness is achieved dynamically—i.e.,

- ✶ by the OS at execution time
- ✶ rather than by the programmers at system configuration time

affects the impact of scaleable timeliness on the scheduler

An OS Architecture Can Be Scaleable In Many Different Respects

❑ Such an OS architecture must be highly *scaleable* with respect to a number of its attributes—

most importantly including

- ✶ functionality
- ✶ performance
- ✶ timeliness
- ✶ predictability
- ✶ decentralization
- ✶ fault tolerance

Perfect Timeliness Is An Ideal But Generally Unrealistic

❑ The ideal case of perfect collective temporal merit

- ✶ every computation always completing execution
- ✶ at an optimum time

is unrealistic in general

❑ Even though the traditional "hard" real-time cases are intended—and commonly imagined—to achieve this ideal,

- ✶ physical laws (especially in decentralized systems)
- ✶ the intrinsic nature of the applications (especially at mission management levels)

generally make it

- ✶ non-cost-effective
- ✶ or even impossible

❑ Thus, the timeliness (temporal merit) of computations and systems is not necessarily assured and known

- ✶ accurately
- ✶ in advance
- ✶ or even at all

Predictability Is The Extent That Timeliness Can Be Estimated Acceptably

- ❑ We consider *predictability* to be the degree to which timeliness can be estimated (in advance) acceptably
- ❑ Predictability applies to any level of a system—e.g.,
  - ∗ individual computations of an application or OS
  - ∗ sets of computations of an application or OS
  - ∗ individual functions or services of an OS
  - ∗ an OS as a whole
  - ∗ a system as a whole
- ❑ The usual practice is for the degree of predictability to be
  - ∗ specified prior to system execution time
  - ∗ attained at system design time
  - ∗ evaluated after system execution time
- ❑ But it is possible for the degree of predictability to be
  - ∗ specified
  - ∗ attained
  - ∗ evaluated

  dynamically (at execution time)

The Degree Cf Predictability Depends On Parameter Knowledge And Scheduler Behavior

- ❑ The degree of predictability of a computation depends on
  - ∗ how well known are all parameter values of
    - ▪ the computation—e.g.,
      - ◊ arrival time
      - ◊ execution duration
    - ▪ and its future execution environment—e.g.,
      - ◊ resource dependencies on
      - ◊ conflicts with

      other computations
  - ∗ how well controlled is the time evolution of the processes which govern the computation's timeliness
    - ▪ particularly the scheduler—

      which must be responsible for scheduling all physical and logical resources, not just processor cycles, in systems needing high
      - ◊ performance of computation completion
      - ◊ timeliness
      - ◊ predictability
    - ▪ e.g., chaotic regimes are a significant risk in highly decentralized schedulers, especially realtime ones

The Degree Of Predictability Is Established According To The Specific Interpretation Of "Acceptably"

- ❑ The degree of predictability is then established according to the application-specific interpretation of "acceptably"—

  e.g., the estimate may be intended to be
  - ∗ extremely precise in certain instances—e.g.,
    - ▪ for certain critical computations or services
    - ▪ under certain critical conditions

    at the expense of being less so in the remainder
  - ∗ versus being
    - ▪ less
    - ▪ but equally

    precise in every instance

Determinism Is The Maximum Case Of Predictability

- ❑ *Deterministic* computation in the realtime context literally means that a computation's, or set of computations', timeliness is known
  - ∗ absolutely
  - ∗ in advance

  i.e.. there is no uncertainty about anything which could affect its timeliness

  (at least barring faults, and preferably within acceptable fault coverage premises)
- ❑ A computation being deterministic does not imply that its timeliness
  - ∗ individually
  - ∗ as a member of a set

  is acceptable, only that it is known

  (although the point of deterministic systems is for them to be known to have acceptable collective timeliness)

digital

Determinism Can Only Be Approached In Practice

❑ There are very few actual realtime applications and systems which (*inherently or forcibly*) meet this determinism criterion of absolute timeliness certainty

❑ Most are subject to some inevitable dynamic fluctuations and variabilities of

- ✱ computation
- ✱ communication

timing, due to factors such as

- ✱ input data arrivals
- ✱ resource dependencies and conflicts
- ✱ overloads
- ✱ hardware and software exceptions

(not to mention faults, errors, and failures outside the presumed coverage)

digital

Predictability Is Often Probabilistic

❑ When the parameters of a computation and its future execution environment are known in the form of random variables,

so that their uncertainty is characterized by probability distribution functions

- ✱ the computation's timeliness may be amenable to stochastic analysis
- ✱ e.g., the probabilities of
  - ▪ execution completion at different times
  - ▪ corresponding temporal merit values

  can be derived for certain situations

❑ But, as with deterministic scheduling, many of the most interesting cases are

- ✱ either known to be analytically intractable
- ✱ or still defy explicit solution

❑ The contexts, and thus approaches, of stochastic scheduling are predominately oriented toward non-realtime objectives, such as makespan or flowtime (throughput measures)

- ✱ which are analytically and computationally easier than stochastic scheduling to meet due times
- ✱ and for which there is greater application demand than from the realtime community

digital

Larger, More Decentralized Realtime Systems Are Generally Non-Stochastically Non-Deterministic

❑ The computation and execution context parameters of many realtime systems,

especially larger, more complex, more decentralized ones,

are *often too asynchronous*—i.e.,

- ✱ intermittent
- ✱ irregular
- ✱ interdependent

to have known, or computationally tractable, probability distribution functions

❑ Thus, these realtime systems must be treated as non-stochastically non-deterministic

- ✱ for which the scheduling technology is still in its infancy
- ✱ although non-realtime
  - ▪ algorithms
  - ▪ languages
  - ▪ models (e.g., Petri Nets)

  commonly take advantage—for simplicity—of making non-stochastically non-deterministic decisions,

  as do an increasing number of realtime algorithms

digital

Predictability Is Most Commonly Expressed As A Least Upper Bound On Timeliness

❑ Predictability may be expressed in a variety of ways—e.g.,

- ✱ an assured upper bound—the most common way

  (a lesser or least upper bound, since any system's timeliness could be said to be predictable by the choice of one high enough)
- ✱ or a probability distribution function of timeliness values
- ✱ or in terms of discontinuous rules which relate various execution contexts to
  - ▪ estimated
  - ▪ bounded
  - ▪ or even specific

  timeliness values—

  those contexts being ones which are most
  - ▪ likely
  - ▪ important
  - ▪ or just readily relatable to timeliness estimations

Predictability Estimates Can Be Made By A Variety Of Techniques

- ❑ Predictability estimates may be made on the basis of
  - ∗ formal analysis
  - ∗ simulation
  - ∗ code examination
  - ∗ empirical measurrment
- ❑ Predictability evaluation
  - ∗ is usually performed by empirical measurement
  - ∗ but in extreme—e.g., asymmtotically deterministic—cases
    - ▪ is unnecessary
    - ▪ because attainment of the specification is assured

      (e.g., through formal synthesis)

OS Predictability Is Traditionally Focused On Timeliness Of Initiating Computations

- ❑ The realtime context and resulting perspective that led commercial realtime OS suppliers and users to traditionally consider
  - ∗ performance of an OS per se

    in terms of the time to initiate computations
  - ∗ and acceptable timeliness of computation completions

    to be primarily an
    - ▪ á priori
    - ▪ application programmer responsibility

  also implies that predictability has been considered likewise
- ❑ Thus, OS predictability has been
  - ∗ specified
  - ∗ attained
  - ∗ evaluated

  predominately
  - ∗ statically—as a design and implementation issue
  - ∗ for the OS as a whole rather than for individual
    - ▪ functions or services
    - ▪ OS or application computations

OS Predictability Is Scaleable To The Extent That It Ranges From No Need To Determinism

- ❑ The predictability of an OS architecture is scaleable to the extent that its instantiations can accommodate
  - ∗ timeliness estimate acceptability ranging
    - ▪ from no need for OS-provided predictability
      - ◊ either no predictability is needed
      - ◊ or all needed predictability is the responsibility of the OS clients (e.g., application programmers)

        and any OS (un)predictability is irrelevant
    - ▪ to asymmtotically deterministic OS-provided predictability,

      under sufficiently certain conditions
  - ∗ at granularities ranging
    - ▪ from individual system and application computations
    - ▪ to individual OS functions or services
    - ▪ to the whole OS
- ❑ This scaleable predictability may be
  - ∗ static—which is less scaleability
  - ∗ dynamic—which is more scaleability
- ❑ We neither quantify nor weight these factors here

Very High Predictability Is Usually Attained With Extraordinary Concepts and Techniques

- ❑ Ubiquitous approaches for attaining very high degrees of system and OS predictability (asymmtotically deterministic)

  of computation completion timeliness
  - ∗ are based on extraordinary concepts and techniques—e.g.,
    - ▪ globally and statically synchronous
      - ◊ computation
      - ◊ i/o
    - ▪ certain formal design and validation methods
  - ∗ that impose extremely high costs of many kinds, including intolerance to any forms of uncertainties
- ❑ Some proponents of those concepts and techniques argue that they are
  - ∗ not only sufficient
  - ∗ but also necessary

  for such high degrees of predictability
- ❑ There is an interesting analogy between predictability and security in this respect

Higher Predictability Does Not Necessarily Imply That All Lower Predictability Needs Can Be Met

- ❑ These concepts and techniques for high predictability are special cases

  which do not scale down to

  - ∗ specifying, attaining, and evaluating
  - ∗ specific or even non-specific

  lesser degrees of predictability

- ❑ This implies that high scaleability of predictability
  - ∗ either must be achieved by new concepts and techniques which do scale well
  - ∗ or is inherently limited

Scaleable Predictability Is About Uncertainties And Is Affected By Scaleable Timeliness And Functionality

- ❑ Predictability is about dealing with uncertainties, so scaleability of predictability must accommodate
  - ∗ varied types and amounts of uncertainties
    - ▪ statically
    - ▪ dynamically
  - ∗ together with varied trade-offs among
    - ▪ predictability
    - ▪ other attributes
    - ▪ and their costs
- ❑ High predictability of computation completion timeliness
  - ∗ depends primarily on the scheduler
  - ∗ and thus the timeliness framework

  so scaleability of predictability is strongly affected by the scaleability of

  - ∗ timeliness
  - ∗ functionality (particularly policy/mechanism separation)

An OS Architecture Can Be Scaleable In Many Different Respects

- ❑ Such an OS architecture must be highly *scaleable* with respect to a number of its attributes—

  most importantly including

  - ∗ functionality
  - ∗ performance
  - ∗ timeliness
  - ∗ predictability
  - ∗ decentralization
  - ∗ fault tolerance

Physical Dispersal Of Processors Is Defined By A Ratio Between State Change And Communication Rates

- ❑ Any particular pair of processors in a system is *physically dispersed* to a degree defined in terms of the ratio between
  - ∗ the rate at which a processor can change state
  - ∗ the rate at which processor state changes can be communicated between them

  (not, as commonly thought, simply to the extent of their separation)

The Magnitude Of This Ratio Is Hardware Dependent

- ❑ This state change/comunication ratio is
  - ✱ relatively small in uniform memory access multiprocessors
  - ✱ somewhat greater in non-uniform memory access (NUMA) multiprocessors—

    i.e, in which memory references take place (typically over a backplane bus) among
    - uniprocessors (with or without local memory)
    - uniform memory access multiprocessors (sometimes called "clusters" in this context)
    - global memory
  - ✱ much greater in multicomputers—processing elements (processor/memory pairs) which intercommunicate by messages over
    - a shared backplane
    - serial bus/ring
    - private links (e.g., meshes)

    (sometimes called NO Remote Memory Access—NORMA—architectures)
  - ✱ greatest in networks and "distributed systems"

The State Change/Communication Ratio Generally Is Dynamic Within Each Range

- ❑ This ratio is in general dynamically variable within each of the above four ranges—e.g., due to
  - ✱ communication queuing in the i/o interconnect architectures
  - ✱ changes in virtual to physical memory mapping in the memory interconnect architectures

Software Is Physically Decentralized To The Degree That Processor Dispersal Is Significant To It

- ❑ Software computations are *physically decentralized* to the degree that

  the state change/comunication ratio which defines the physical dispersal of processors
  - ✱ is significant to
  - ✱ i.e., must be explicitly recognized and accommodated by

  those computations themselves
- ❑ We regard this significance as qualitative and do not quantify it
- ❑ Physical centralization is one end point in the dimension of physical decentralization
- ❑ (Software is also *logically decentralized* to different degrees, which we will not address here)

Physical Dispersal Has Various Fundamental Effects Which Are Significant To Decentralized Software

- ❑ The significance of the processor state change/comunication ratio is manifest in the software's need to explicitly recognize and accommodate aspects of

  locality of references in space and time
  - ✱ the binding of computations'
    - code segments
    - current execution points
    - data

    to pro[illegible]ors
  - ✱ the
    - identities
    - physical locations

    of the processors
  - ✱ the
    - magnitudes
    - uniformity
    - variability

    of the interprocessor communication times
    - whether memory (in the case of multiprocessors)
    - or i/o (in the cases of multicomputers, networks, and distributed systems)

Only Node-Local Computations Are Centralized

- □ The only computations that are centralized (i.e., to which the processor state change/comunication ratio is insignificant)

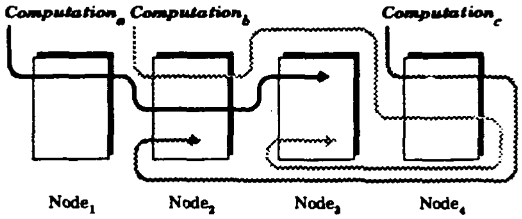
  are those which are entirely local to a *node*

  - ∗ either a uniprocessor—i.e., one processor/memory pair
  - ∗ or multiple processors having negligible physical dispersal—

    i.e., a multiprocessor with only globally shared, uniform access time, memory

In Multinode Computer Systems Some Computations Span Multiple Nodes

- □ In computer systems that have a multiplicity of nodes, there must be some computations that
  - ∗ span multiple nodes—i.e., are *trans-node*



  - ∗ and thus are necessarily physically decentralized to some degree

- □ The least decentralized that multinode computations can be is on a NUMA multiprocessor
  - ∗ by definition

    (i.e., differentiating it from a UMA multiprocessor, which is single-node)

  - ∗ its state change/comunication ratio has unavoidably lower bounded significance to all trans-node computations—

    e.g., on locality of code and data references in space and time (and thus on performance at least)

Computations At Different Levels Of A System Are Generally Decentralized To Different Degrees

- □ Computations exist at different levels of the system
  - ∗ from the applications
  - ∗ down to the OS and kernel
  - ∗ (and even the processor interconnect hardware can be thought of as comprising computations)
- □ The physical decentralization of computations can differ at different levels of the system

  (physical decentralization of the processor interconnect hardware is always maximum)

An OS Is Decentralized To The Extent That All Its Computations Are

- □ A centralized kernel or OS (or any other level in the system) is one which has only centralized computations and services—

  which implies that

  - ∗ its computations and services are confined entirely to a single node
  - ∗ any accommodation or exploitation of physical dispersal must be performed at one or more higher levels in the system

- □ A kernel or an OS (or any other level in the system) is decentralized to the extent that
  - ∗ each of its computations and services
  - ∗ is decentralized

    and thus trans-node

A Decentralized OS Generally Is Not Suitable For Lower Or Higher Physical Dispersal Than Intended

❑ A kernel or an OS (or any other level in the system) which is decentralized to any given degree

will not necessarily be suitable for a different

* ✱ lower
* ✱ higher

physical dispersal than it was intended for

An OS Intended For Lower Physical Dispersal Generally Will Not Function Correctly With Higher

❑ A kernel or OS (or any other level in the system) intended for lower physical dispersal generally will not function correctly with higher physical dispersal,

due to its lack of capability for decentralization (accommodating effects of the state change/comunication ratio)—e.g.,

✱ a centralized OS generally will not work for a NUMA multiprocessor—

e.g., because its centralized virtual memory management cannot handle the non-locality of concurrent references among "clusters"

✱ an OS for a NUMA multiprocessor generally will not work for a "distributed" (NORMA) system—

e.g., because of the absence of coherent shared global state which it depends on,

such as for intercomputation communication and synchronization

An OS Intended For Higher Physical Dispersal Generally Is Not Cost-Effective With Lower

❑ A kernel or OS (or any other level in the system) intended for higher physical dispersal generally is not cost-effective with lower physical dispersal,

due to the execution overhead of decentralization (accommodating effects of the state change/comunication ratio)—e.g.,

✱ any multiprocessor OS has unnecessary overhead on a uniprocessor—

e.g., because of its locks

✱ a NUMA multiprocessor OS has even more unnecessary overhead on a single-node machine—

e.g., because of its more complex virtual memory management

✱ a "distributed" OS may have unnecessary overhead on a

- NUMA multiprocessor
- single-node machine

because of its facilities (e.g., for intercomputation communication) to overcome the absence of coherent shared global state

Decentralization Is Scaleable To The Extent That It Is Independent Of The Magnitude Of Dispersal

❑ The decentralization of a computation is scaleable to the extent that the

✱ the significance to that computation of physical dispersal (the state change/comunication ratio)

✱ is independent of the magnitude of the physical dispersal

❑ The decentralization of an

* ✱ OS service
* ✱ OS
* ✱ or any other level in the system

is scaleable to the extent that the decentralization of each of its computations is scaleable

❑ An OS (or any other level in the system) which has maximally scaleable decentralization is entirely independent of physical dispersal—

i.e., can operate correctly on any

* ✱ single-node
* ✱ multinode

architecture

(cf. delay-insensitive logic)

The Decentralization Of Computations At Each Level Is A Fundamental Multinode Architecture Decision

□ A fundamental multinode system architecture decision is the degree of physical decentralization

- ✱ not just at each level of the system
- ✱ but also for the various computations at each level

One End Of The Multinode Architecture Spectrum Is Highly Physically Decentralized At Every Level

□ One end of the multinode system architecture spectrum reflects the processor dispersal up to the—thus highly physically decentralized—application level(s)

to achieve efficiency benefits from

- ✱ the programming and execution structures of the system being relatively congruent with that of the application—e.g.,
  - M-ary N-cube architectures
  - and message-based OS's

  are a good match for the computational structure of certain physical science applications
- ✱ avoiding overhead incurred by virtualizing away the the physical dispersal

High Decentralization At All Levels Of The System Has Been Popular For Supercomputers And Realtime

□ This end of the spectrum is historically—but diminishingly—the choice for multinode

- ✱ supercomputers
- ✱ realtime computers

because the users of each have tended to

- ✱ trade off cost-effectiveness for maximum performance
- ✱ be less concerned with
  - legacy software
  - costs of learning and tools for decentralization

The Other End Of The Multinode Architecture Spectrum Is Highly Physically Centralized At Every Level

□ The other end of the multinode system architecture spectrum has

- ✱ as many computations
- ✱ as physically centralized as possible

□ The goal of this is to minimize the impact of physical dispersal on software costs—e.g., by

- staying closer to familiar centralized programming techniques and tools
- preserving legacy software
- being independent of the physical dispersal aspects of different multinode architectures

□ The approach is

- ✱ to create a virtual system for the maximum number of the higher levels

  which is
  - as centralized as possible
  - given the processor physical dispersal
- ✱ by being highly decentralized at
  - the minimum number
  - of the lowest level(s)

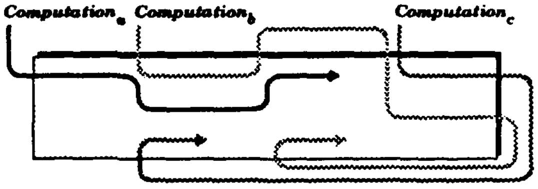
Lower Levels Can Create A Virtual NUMA System So That Higher Levels Can Be More Centralized

❑ Trans-node computations at higher levels can be more physically centralized

if the trans-node computations at one or more levels below it

∗ create a virtual more centralized system

∗ by being highly physically decentralized—e.g., providing a high degree of node transparency



Single NUMA Multiprocessor

❑ The most centralized virtual machine possible in a multinode system is a NUMA multiprocessor at all levels

∗ virtualization cannot reduce the fixed (or lower bounded) state change/comunication ratios

∗ these ratios have unavoidably lower bounded significance to all trans-node computations—

e.g., locality of reference in space and time

A Virtual NUMA Multiprocessor Can Be Created By The Processor Interconnect Or Kernel/OS Levels

❑ The lowest system level which can create this virtual NUMA multiprocessor is the processor interconnect hardware—

cf. the KSR-1, and numerous *distributed shared memory* research projects

∗ minimizes multinode impact on all software from the OS kernel up

∗ requires innovative, non-standard, expensive processor interconnect hardware

❑ Virtualization assistance may be provided by the OS kernel to simplify the interconnect hardware—

cf. SGI's rumored forthcoming multinode product, and numerous other distributed shared memory research projects

❑ Given conventional processor interconnect hardware that doesn't virtualize the nodes,

the kernel and OS are the lowest levels which can do so

∗ cf. the OSF/RI version of Mach 3 and OSF-1 (OSF-1/AD) that provides NUMA virtualization on Intel's NORMA Paragon hypercube

∗ (not to be confused with the version of Mach 3 that CMU modified to run on NUMA multiprocessors)

A Virtual NUMA Multiprocessor Is Often Desired For Extant OS And Application Layers

❑ Presently, it is most frequently desired that multinode systems have minimum impact on the extant

∗ OS

∗ as well as applications

❑ This implies that one or more levels of computation between the hardware and the applications must

∗ be highly decentralized

∗ and provide the desired degree of virtualization (e.g., a NUMA multiprocessor)

❑ Some of this virtualization may be provided by intermediate levels such as

∗ a distributed execution environment (e.g., DCE)

∗ an object-oriented execution environment (e.g., OMA-based)

❑ But such intermediate levels typically

∗ do not have direct access to kernel and OS level resources

∗ only via conventional centralized OS services, which limits their degree of

- decentralization
- timeliness

The Degree Of Decentralization Need Not Change Monotonically By System Level

❑ The degree to which computations are physically decentralized need not change monotonically by system level—

e.g., physical decentralization is commonly

∗ low at the OS level

to allow the use of extant node OS's which were not intended to perform trans-node management of resources other than for networking

∗ high at an intermediate distributed execution environment (e.g., DCE) level

to reduce the trans-node resource management obligations of the

- application programs above
- node OS's below

∗ moderate at the application software level

- to reduce the trans-node resource management obligations (thus costs) of those programs
- while retaining ability to sufficiently manage and exploit the system's structure

There Are Needs To Bypass Virtualization In Multinode Systems

❑ The programmers at a physically centralized level occasionally need to

- ✱ bypass some virtualization
- ✱ and perhaps also employ some decentralized computation

e.g.,

- ✱ when they desire to see or control some software/ ˈhːrdware binding for
  - ▪ performance—due to node locality of execution and data access
  - ▪ fault tolerance—by partitioning and replication
- ✱ in the case of certain service outages where
  - ▪ application-specific recourse must be taken
  - ▪ or the end-to-end argument applies

An OS Architecture Can Be Scaleable In Many Different Respects

❑ Such an OS architecture must be highly *scaleable* with respect to a number of its attributes—

most importantly including

- ✱ functionality
- ✱ performance
- ✱ timeliness
- ✱ predictability
- ✱ decentralization
- ✱ fault tolerance

OS Fault Tolerance Is Scaleable To The Extent That Specific Kinds And Degrees Can Be Provided

❑ Fault tolerance is the extent to which a system

- ✱ either exhibits a well-defined failure behavior when elements fail
- ✱ or masks element failures (continues to provide service) to its users

❑ In most applications, temporary errant behavior or service unavailability is acceptable;

in many others—especially realtime ones

- ✱ one or both kinds of fault tolerance are required
- ✱ to various degrees
- ✱ under various circumstances

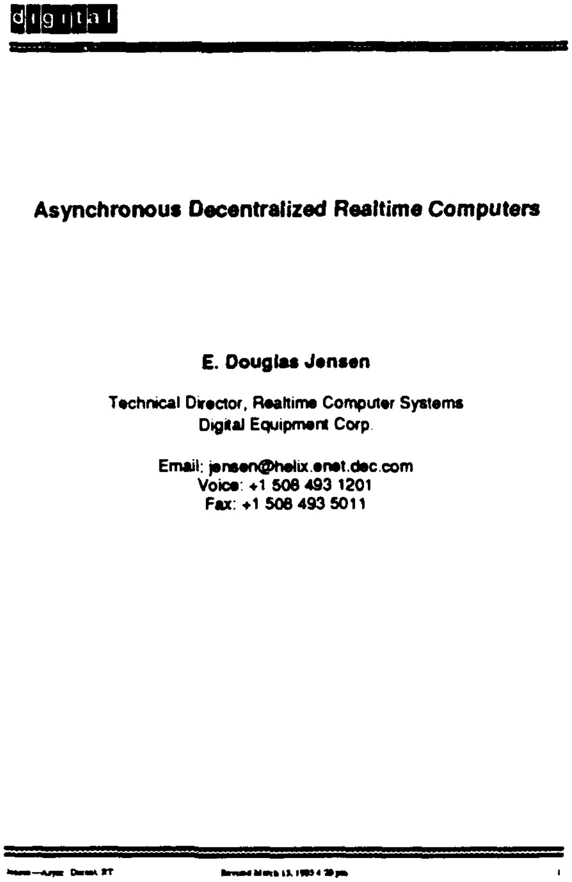
❑ An OS architecture's fault tolerance is scaleable to the extent that its instantiations can exhibit the kind and degree of fault tolerance desired for a particular

- ✱ system
- ✱ service
- ✱ computation
- ✱ circumstance

stuff about how to have scaleable fault tolerance

❑

More...

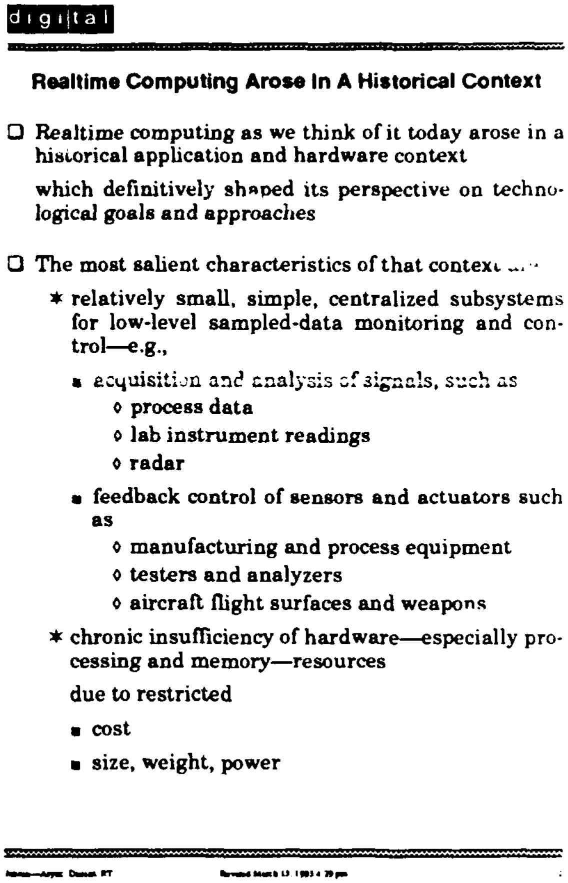


Asynchronous Decentralized Realtime Computers

E. Douglas Jensen

Technical Director, Realtime Computer Systems
Digital Equipment Corp.

Email: jensen@helix.enet.dec.com
Voice: +1 508 493 1201
Fax: +1 508 493 5011



Realtime Computing Arose In A Historical Context

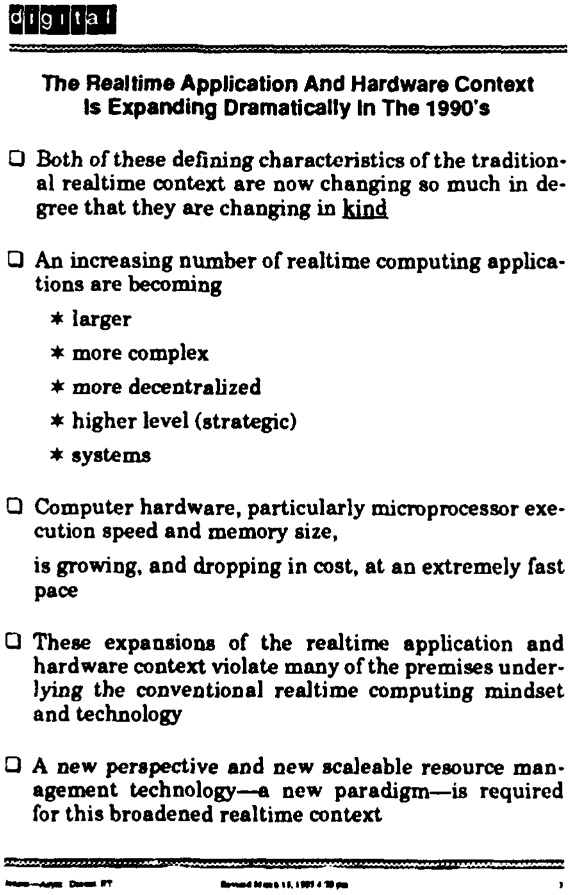
❑ Realtime computing as we think of it today arose in a historical application and hardware context

which definitively shaped its perspective on technological goals and approaches

❑ The most salient characteristics of that context are

- ✱ relatively small, simple, centralized subsystems for low-level sampled-data monitoring and control—e.g.,
  - ▪ acquisition and analysis of signals, such as
    - ◊ process data
    - ◊ lab instrument readings
    - ◊ radar
  - ▪ feedback control of sensors and actuators such as
    - ◊ manufacturing and process equipment
    - ◊ testers and analyzers
    - ◊ aircraft flight surfaces and weapons
- ✱ chronic insufficiency of hardware—especially processing and memory—resources

  due to restricted
  - ▪ cost
  - ▪ size, weight, power



The Realtime Application And Hardware Context Is Expanding Dramatically In The 1990's

❑ Both of these defining characteristics of the traditional realtime context are now changing so much in degree that they are changing in <u>kind</u>

❑ An increasing number of realtime computing applications are becoming

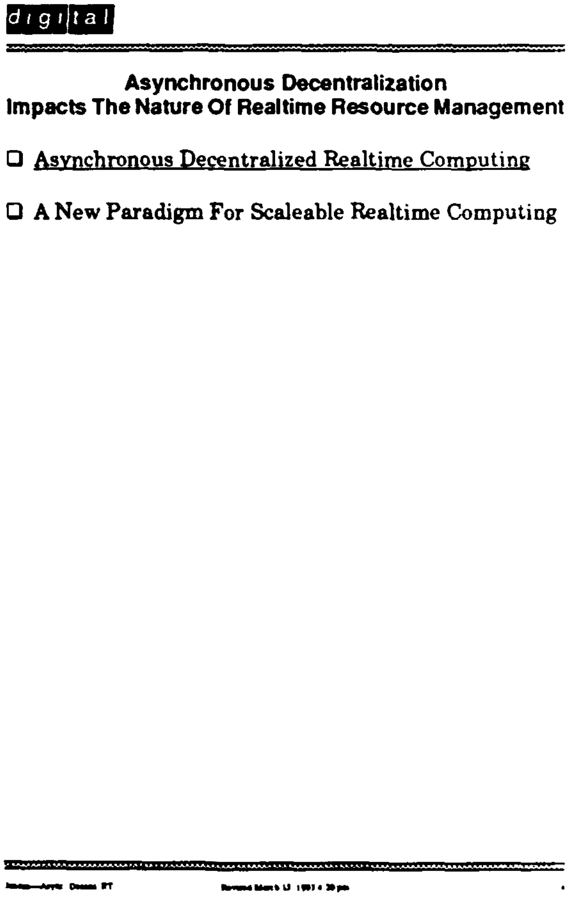
- ✱ larger
- ✱ more complex
- ✱ more decentralized
- ✱ higher level (strategic)
- ✱ systems

❑ Computer hardware, particularly microprocessor execution speed and memory size,

is growing, and dropping in cost, at an extremely fast pace

❑ These expansions of the realtime application and hardware context violate many of the premises underlying the conventional realtime computing mindset and technology

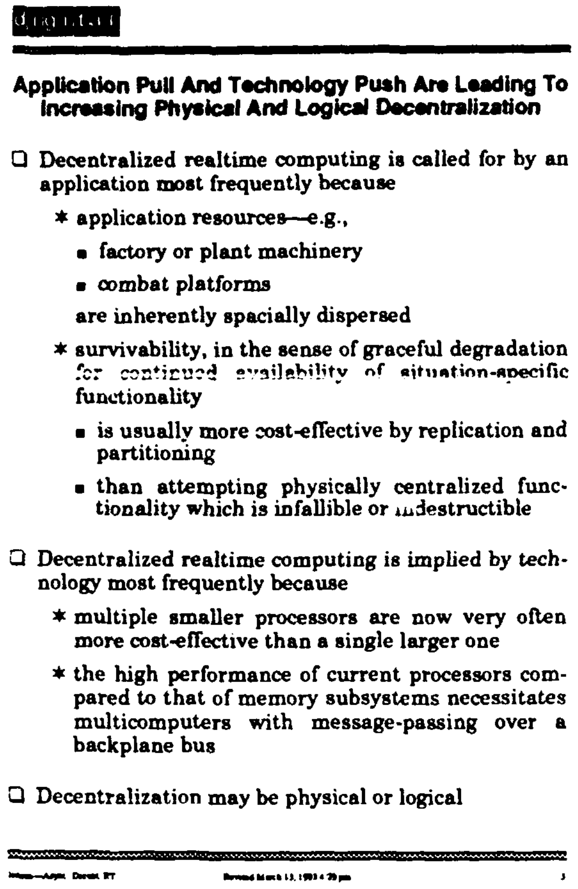
❑ A new perspective and new scaleable resource management technology—a new paradigm—is required for this broadened realtime context



Asynchronous Decentralization Impacts The Nature Of Realtime Resource Management

❑ <u>Asynchronous Decentralized Realtime Computing</u>

❑ A New Paradigm For Scaleable Realtime Computing



Application Pull And Technology Push Are Leading To Increasing Physical And Logical Decentralization

❑ Decentralized realtime computing is called for by an application most frequently because

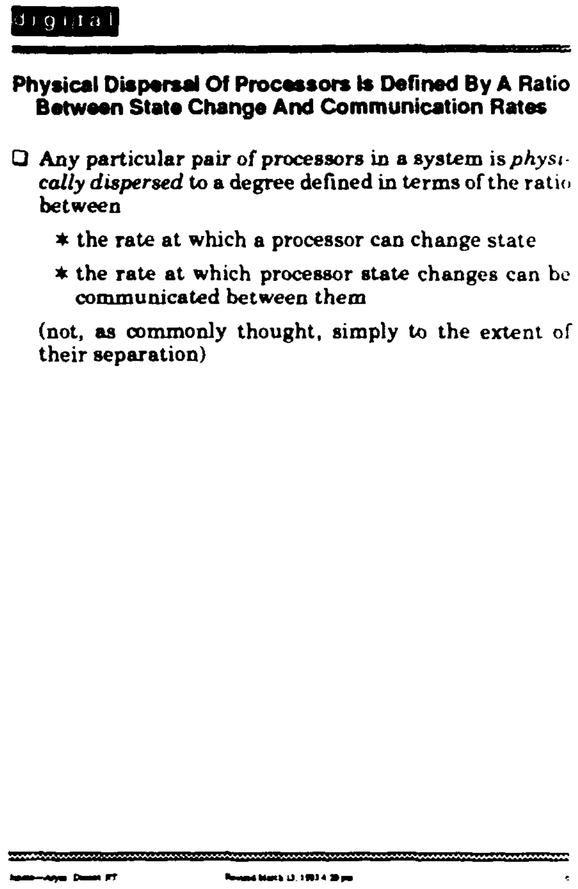
- ✱ application resources—e.g.,
  - factory or plant machinery
  - combat platforms

  are inherently spacially dispersed
- ✱ survivability, in the sense of graceful degradation for continued availability of situation-specific functionality
  - is usually more cost-effective by replication and partitioning
  - than attempting physically centralized functionality which is infallible or indestructible

❑ Decentralized realtime computing is implied by technology most frequently because

- ✱ multiple smaller processors are now very often more cost-effective than a single larger one
- ✱ the high performance of current processors compared to that of memory subsystems necessitates multicomputers with message-passing over a backplane bus

❑ Decentralization may be physical or logical

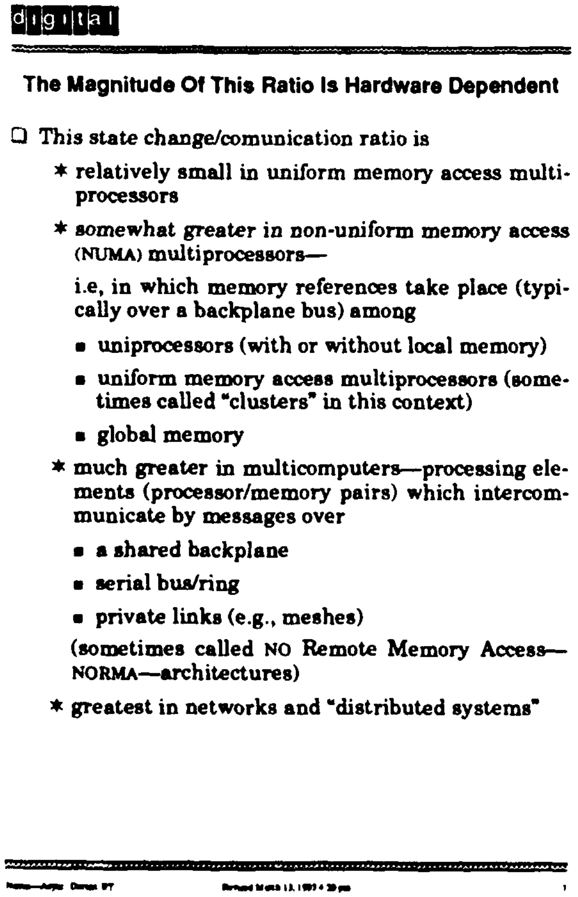


Physical Dispersal Of Processors Is Defined By A Ratio Between State Change And Communication Rates

❑ Any particular pair of processors in a system is *physically dispersed* to a degree defined in terms of the ratio between

- ✱ the rate at which a processor can change state
- ✱ the rate at which processor state changes can be communicated between them

(not, as commonly thought, simply to the extent of their separation)



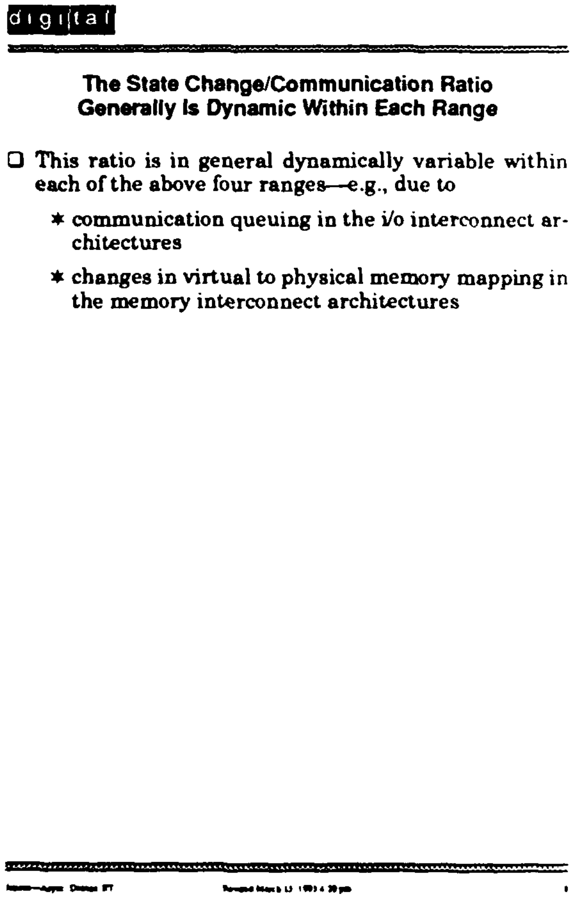
The Magnitude Of This Ratio Is Hardware Dependent

❑ This state change/comunication ratio is

- ✱ relatively small in uniform memory access multiprocessors
- ✱ somewhat greater in non-uniform memory access (NUMA) multiprocessors—

  i.e, in which memory references take place (typically over a backplane bus) among
  - uniprocessors (with or without local memory)
  - uniform memory access multiprocessors (sometimes called "clusters" in this context)
  - global memory
- ✱ much greater in multicomputers—processing elements (processor/memory pairs) which intercommunicate by messages over
  - a shared backplane
  - serial bus/ring
  - private links (e.g., meshes)

  (sometimes called NO Remote Memory Access—NORMA—architectures)
- ✱ greatest in networks and "distributed systems"



The State Change/Communication Ratio Generally Is Dynamic Within Each Range

❑ This ratio is in general dynamically variable within each of the above four ranges—e.g., due to

- ✱ communication queuing in the i/o interconnect architectures
- ✱ changes in virtual to physical memory mapping in the memory interconnect architectures

digital

Software Is Physically Decentralized To The Degree That Processor Dispersal Is Significant To It

❑ Software computations are *physically decentralized* to the degree that

the state change/comunication ratio which defines the physical dispersal of processors

- ✱ is significant to
- ✱ i.e., must be explicitly recognized and accommodated by

those computations themselves

❑ We regard this significance as qualitative and do not quantify it

❑ Physical centralization is one end point in the dimension of physical decentralization

❑ (Software is also *logically decentralized* to different degrees, which we will address later)

digital

Physical Dispersal Has Various Fundamental Effects Which Are Significant To Decentralized Software

❑ The significance of the processor state change/comunication ratio is manifest in the software's need to explicitly recognize and accommodate aspects of

locality of references in space and time

- ✱ the binding of computations'
  - ■ code segments
  - ■ current execution points
  - ■ data

  to processors
- ✱ the
  - ■ identities
  - ■ physical locations

  of the processors
- ✱ the
  - ■ magnitudes
  - ■ uniformity
  - ■ variability

  of the interprocessor communication times
  - ■ whether memory (in the case of multiprocessors)
  - ■ or i/o (in the cases of multicomputers, networks, and distributed systems)

digital

Only Node-Local Computations Are Centralized

❑ The only computations that are centralized (i.e., to which the processor state change/comunication ratio is insignificant)

are those which are entirely local to a *node*

- ✱ either a uniprocessor—i.e., one processor/memory pair
- ✱ or multiple processors having negligible physical dispersal—

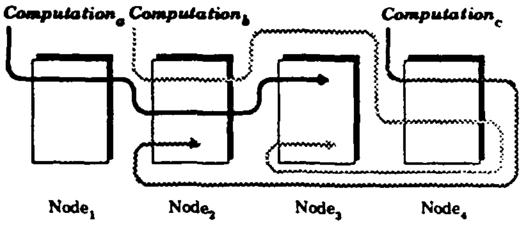
  i.e., a multiprocessor with only globally shared, uniform access time, memory

digital

In Multinode Computer Systems Some Computations Span Multiple Nodes

❑ In computer systems that have a multiplicity of nodes, there must be some computations that

- ✱ span multiple nodes—i.e., are *trans-node*



- ✱ and thus are necessarily physically decentralized to some degree

❑ The least decentralized that multinode computations can be is on a NUMA multiprocessor

- ✱ by definition

  (i.e., differentiating it from a UMA multiprocessor, which is single-node)
- ✱ its state change/comunication ratio has unavoidably lower bounded significance to all trans-node computations—

  e.g., on locality of code and data references in space and time (and thus on performance at least)

Computations At Different Levels Of A System Are Generally Decentralized To Different Degrees

❑ Computations exist at different levels of the system

- ∗ from the applications
- ∗ down to the OS and kernel
- ∗ (and even the processor interconnect hardware can be thought of as comprising computations)

❑ The physical decentralization of computations can differ at different levels of the system

(physical decentralization of the processor interconnect hardware is always maximum)

An OS Is Decentralized To The Extent That All Its Computations Are

❑ A centralized kernel or OS (or any other level in the system) is one which has only centralized computations and services—

which implies that

- ∗ its computations and services are confined entirely to a single node
- ∗ any accommodation or exploitation of physical dispersal must be performed at one or more higher levels in the system

❑ A kernel or an OS (or any other level in the system) is decentralized to the extent that

- ∗ each of its computations and services
- ∗ is decentralized

and thus trans-node

A Decentralized OS Generally Is Not Suitable For Lower Or Higher Physical Dispersal Than Intended

❑ A kernel or an OS (or any other level in the system) which is decentralized to any given degree

will not necessarily be suitable for a different

- ∗ lower
- ∗ higher

physical dispersal than it was intended for

An OS Intended For Lower Physical Dispersal Generally Will Not Function Correctly With Higher

❑ A kernel or OS (or any other level in the system) intended for lower physical dispersal generally will not function correctly with higher physical dispersal,

due to its lack of capability for decentralization (accommodating effects of the state change/comunication ratio)—e.g.,

- ∗ a centralized OS generally will not work for a NUMA multiprocessor—

  e.g., because its centralized virtual memory management cannot handle the non-locality of concurrent references among "clusters"

- ∗ an OS for a NUMA multiprocessor generally will not work for a "distributed" (NORMA) system—

  e.g., because of the absence of coherent shared global state which it depends on,

  such as for intercomputation communication and synchronization

An OS Intended For Higher Physical Dispersal Generally Is Not Cost-Effective With Lower

❑ A kernel or OS (or any other level in the system) intended for higher physical dispersal generally is not cost-effective with lower physical dispersal,

due to the execution overhead of decentralization (accommodating effects of the state change/comunication ratio)—e.g.,

- ∗ any multiprocessor OS has unnecessary overhead or a uniprocessor—

  e.g., because of its locks

- ∗ a NUMA multiprocessor OS has even more unnecessary overhead on a single-node machine—

  e.g., because of its more complex virtual memory management

- ∗ a "distributed" OS may have unnecessary overhead on a
  - ▪ NUMA multiprocessor
  - ▪ single-node machine

  because of its facilities (e.g., for intercomputation communication) to overcome the absence of coherent shared global state

Decentralization Is Scaleable To The Extent That It Is Independent Of The Magnitude Of Dispersal

❑ The decentralization of a computation is scaleable to the extent that the

- ∗ the significance to that computation of physical dispersal (the state change/comunication ratio)
- ∗ is independent of the magnitude of the physical dispersal

❑ The decentralization of an

- ∗ OS service
- ∗ OS
- ∗ or any other level in the system

is scaleable to the extent that the decentralization of each of its computations is scaleable

❑ An OS (or any other level in the system) which has maximally scaleable decentralization is entirely independent of physical dispersal—

i.e., can operate correctly on any

- ∗ single-node
- ∗ multinode

architecture

(cf. delay-insensitive logic)

The Decentralization Of Computations At Each Level Is A Fundamental Multinode Architecture Decision

❑ A fundamental multinode system architecture decision is the degree of physical decentralization

- ∗ not just at each level of the system
- ∗ but also for the various computations at each level

One End Of The Multinode Architecture Spectrum Is Highly Physically Decentralized At Every Level

❑ One end of the multinode system architecture spectrum reflects the processor dispersal up to the—thus highly physically decentralized—application level(s)

to achieve efficiency benefits from

- ∗ the programming and execution structures of the system being relatively congruent with that of the application—e.g.,
  - ▪ M-ary N-cube architectures
  - ▪ and message-based OS's

  are a good match for the computational structure of certain physical science applications

- ∗ avoiding overhead incurred by virtualizing away the the physical dispersal

High Decentralization At All Levels Of The System Has Been Popular For Supercomputers And Realtime

❑ This end of the spectrum is historically—but diminishingly—the choice for multinode

- ∗ supercomputers
- ∗ realtime computers

because the users of each have tended to

- ∗ trade off cost-effectiveness for maximum performance
- ∗ be less concerned with
  - ▪ legacy software
  - ▪ costs of learning and tools for decentralization

The Other End Of The Multinode Architecture Spectrum Is Highly Physically Centralized At Every Level

❑ The other end of the multinode system architecture spectrum has

- ∗ as many computations
- ∗ as physically centralized as possible

❑ The goal of this is to minimize the impact of physical dispersal on software costs—e.g., by

- ▪ staying closer to familiar centralized programming techniques and tools
- ▪ preserving legacy software
- ▪ being independent of the physical dispersal aspects of different multinode architectures

❑ The approach is

- ∗ to create a virtual system for the maximum number of the higher levels

  which is

  - ▪ as centralized as possible
  - ▪ given the processor physical dispersal
- ∗ by being highly decentralized at
  - ▪ the minimum number
  - ▪ of the lowest level(s)

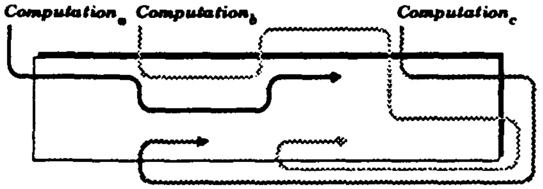
Lower Levels Can Create A Virtual NUMA System So That Higher Levels Can Be More Centralized

❑ Trans-node computations at higher levels can be more physically centralized

if the trans-node computations at one or more levels below it

- ∗ create a virtual more centralized system
- ∗ by being highly physically decentralized—e.g.,

  providing a high degree of node transparency



Single NUMA Multiprocessor

❑ The most centralized virtual machine possible in a multinode system is a NUMA multiprocessor at all levels

- ∗ virtualization cannot reduce the fixed (or lower bounded) state change/comunication ratios
- ∗ these ratios have unavoidably lower bounded significance to all trans-node computations—

  e.g., locality of reference in space and time

A Virtual NUMA Multiprocessor Can Be Created By The Processor Interconnect Or Kernel/OS Levels

❑ The lowest system level which can create this virtual NUMA multiprocessor is the processor interconnect hardware—

cf. the KSR-1, and numerous *distributed shared memory* research projects

- ∗ minimizes multinode impact on all software from the OS kernel up
- ∗ requires innovative, non-standard, expensive processor interconnect hardware

❑ Virtualization assistance may be provided by the OS kernel to simplify the interconnect hardware—

cf. SGI's rumored forthcoming multinode product, and numerous other distributed shared memory research projects

❑ Given conventional processor interconnect hardware that doesn't virtualize the nodes,

the kernel and OS are the lowest levels which can do so

- ∗ cf. the OSF/RI version of Mach 3 and OSF-1 (OSF-1/AD) that provides NUMA virtualization on Intel's NORMA Paragon hypercube
- ∗ (not to be confused with the version of Mach 3 that CMU modified to run on NUMA multiprocessors)

A Virtual NUMA Multiprocessor Is Often Desired For Extant OS And Application Layers

- ❑ Presently, it is most frequently desired that multinode systems have minimum impact on the extant
  - ∗ OS
  - ∗ as well as applications
- ❑ This implies that one or more levels of computation between the hardware and the applications must
  - ∗ be highly decentralized
  - ∗ and provide the desired degree of virtualization (e.g., a NUMA multiprocessor)
- ❑ Some of this virtualization may be provided by intermediate levels such as
  - ∗ a distributed execution environment (e.g., DCE)
  - ∗ an object-oriented execution environment (e.g., OMA-based)
- ❑ But such intermediate levels typically
  - ∗ do not have direct access to kernel and OS level resources
  - ∗ only via conventional centralized OS services, which limits their degree of
    - ■ decentralization
    - ■ timeliness

The Degree Of Decentralization Need Not Change Monotonically By System Level

- ❑ The degree to which computations are physically decentralized need not change monotonically by system level—

  e.g., physical decentralization is commonly
  - ∗ low at the OS level

    to allow the use of extant node OS's which were not intended to perform trans-node management of resources other than for networking
  - ∗ high at an intermediate distributed execution environment (e.g., DCE) level

    to reduce the trans-node resource management obligations of the
    - ■ application programs above
    - ■ node OS's below
  - ∗ moderate at the application software level
    - ■ to reduce the trans-node resource management obligations (thus costs) of those programs
    - ■ while retaining ability to sufficiently manage and exploit the system's structure

There Are Needs To Bypass Virtualization In Multinode Systems

- ❑ The programmers at a physically centralized level occasionally need to
  - ∗ bypass some virtualization
  - ∗ and perhaps also employ some decentralized computation

  e.g.,
  - ∗ when they desire to see or control some software/hardware binding for
    - ■ performance—due to node locality of execution and data access
    - ■ fault tolerance—by partitioning and replication
  - ∗ in the case of certain service outages where
    - ■ application-specific recourse must be taken
    - ■ or the end-to-end argument applies

Logical Decentralization Relates To The Form Of Multilateral Activity

- ❑ We regard a computation's *logical decentralization* to be the degree to which it is performed *multilaterally*, determined by
  - ∗ *consentaneity*—the extent to which the participating entities must contribute to the computation before it is complete
  - ∗ *equipollence*—the functional parity of the participating entities
  - ∗ the number of participating entities
- ❑ A quintessential form of utmost logical decentralization is negotiated consensus among autonomous peers
- ❑ Intermediate forms of logical decentralization are exemplified by
  - ∗ succession—where all activities of a computation are performed for a period of time by one entity, and then by another, in some serial sequence
  - ∗ partitioning—where each entity performs a different activity of the computation, whether consecutively or concurrently

Physical and Logical Decentralization Tend To Interact

❑ High degrees of physical decentralization at some level

imply significant logical decentralization of at least some resource management at that level is valuable or essential

❑ High degrees of logical decentralization at some level, such as the application, in a multinode context

imply high degrees of physical decentralization is present at that or lower levels

❑ High degrees of both logical and physical decentralization can easily have extremely complex dynamics which result in chaotic behavior

- ✶ avoiding chaos while maintaining high performance and adaptivity in such systems having many degrees of freedom requires sophisticated control techniques which are as yet nascent
- ✶ the very strong coupling sometimes employed to construct highly predictable realtime computer systems for low-level applications (e.g., MARS)
  - ▪ that are both logically and physically decentralized to significant degrees
  - ▪ at the expense of adaptability

  is sufficient but not necessary for the avoidance of chaotic behavior

Decentralized Realtime Applications Considered Here Are For Mission Management

❑ Some (including the earliest) modestly decentralized realtime applications are

- ✶ low-level, synchronous, sampled data communication, monitoring, and processing
- ✶ subsystems

e.g., process control, sonar signal processing

❑ But the decentralized realtime applications of interest here are those strategic ones now emerging for the purpose of managing the entire system's mission—

e.g., coordination of multiple entities which are

- ✶ manufacturing a vehicle
- ✶ repairing a damaged reactor
- ✶ controlling air or rail traffic
- ✶ conducting a combat engagement

❑ Decentralized mission management applications

- ✶ are in addition to
- ✶ employ and control

the constituent lower-level (centralized and decentralized) realtime subsystems

Decentralized Realtime Mission Management Systems Generally Are Subject To Extraordinary Uncertainty

❑ Realtime mission management that is highly physically and logically decentralized is distinctive in the extent to which it is subject to extraordinary execution-time uncertainties at the application levels

❑ The computations inevitably

- ✶ are asynchronous (mutually, globally)—e.g., event driven, aperiodic
- ✶ have dynamic dependencies—e.g., resource conflicts, precedence constraints
- ✶ are co-evolving—each computation's behavior depends on that of others
- ✶ often constitute an overload
- ✶ permit little if any downtime for repairs or reconfiguration

❑ Computing system physical distribution per se also generally introduces considerable additional uncertainties—

e.g., variable, unknown communication latencies

Decentralized Realtime Mission Management Systems Require High Dependability

❑ The degree of mission success is determined by the extent to which the system can be depended upon to provide sufficient

- ✶ timeliness
- ✶ survivability
- ✶ safety and security

❑ The dependability of lower layer subsystems—the goal of traditional realtime computing—may be

- ✶ either necessary for mission-critical functions

  (e.g., digital avionics flight control keeping the aircraft aloft)
- ✶ or part of the uncertainty to be tolerated at the system and mission layers

  (e.g., communications, weapons in various states of usability)

but it is not sufficient

(e.g., a flying aircraft which cannot perform its mission is wasting resources and creating risks)

Resolving Realtime Dependability And Uncertainty Is Often Beyond The Capability Of System Operators

❑ In decentralized mission management systems, a major challenge is simultaneously

- * achieving sufficient dependability
- * accommodating execution-time uncertainties

❑ Virtually all realtime reconciliation of uncertainty and dependability at the system and mission levels has historically been based solely on the talent and expertise of the system's human operators—e.g.,

- * in the control rooms of factories and plants
- * in the cockpits of aircraft

❑ Increasingly, the

- * complexity and pace of the systems' missions
- * the number, complexity, and distribution of their resources

cause cognitive overload—

which requires that these operators receive more support in this respect from the computing system itself

❑ Such support is beginning to appear at the application levels in a variety of non-realtime computing systems,

but realtime constraints require it at the system software levels as well

Decentralized Realtime Mission Management Requires Best-Effort Realtime Resource Management

❑ In decentralized mission management systems, both the application and computer system software (e.g., OS, DCE) must make an on-line *best effort* to

- * accommodate dynamic and non-deterministic
  - ▪ external (application environment)
  - ▪ internal (system resource)

  conditions
- * in a robust, adaptable way so as to undertake that
  - ▪ as many as possible
  - ▪ of the most important computations
  - ▪ are as acceptable, in the time and other domains, to the application as possible

❑ Best-effort resource management is generally heuristic

- * the use of heuristics for non-realtime computing is
  - ▪ common in applications (most conspicuously in artificial intelligence, pattern recognition)
  - ▪ less familiar in system software
- * but heuristics have been foreign to realtime resource management—focused on static determinism—until Jensen's Archons project and Alpha kernel for mission management

Best-Effort Resource Management Involves Trade-Offs Of Risk And Situational Coverage

❑ Best-effort realtime resource management involves trade-offs of risk and situational coverage

- * best-effort on-line realtime scheduling heuristics currently offer
  - ▪ empirically-based high confidence that acceptable computational timeliness will be achieved over a broad range of realistic conditions
  - ▪ but no, or low, formal bounds on guaranteed best case timeliness

  (as is necessarily the case for human operators)
- * traditional off-line "hard" realtime scheduling algorithms provide
  - ▪ formal guarantees of optimum computational timing under extremely restricted conditions
  - ▪ but behavior which is unknown, or known to be pathologically wrong, outside those conditions

❑ Examples of realtime applications which seem to call naturally for each of these extremes come immediately to mind—

but beware of the human trait to miscalculate risks

- * people erroneously undervalue the reduction of risk
- * in comparison to the elimination of risk

Decentralized Realtime Mission Management Calls For A New Paradigm

❑ Conventional realtime resource management attitudes and technology do not permit such application-specific trade-offs between

- * situational coverage
- * optimality and predictability

❑ A new, more

- * general
- * scaleable

realtime computing paradigm is needed to better accommodate asynchronous decentralized realtime computing systems

❑ Paradigm shifts are rather uncommon in computing—e.g.,

- * parallel processing
- * data flow models

and virtually unprecedented in realtime computing

An Analogy Can Be Drawn From Gravity

- ❑ Prior to Newton's law, people felt that they understood gravity, as evidenced by the fact that they could take it into account in building acceptably stable mechanical constructions
- ❑ But some scientists were dissatisfied with this understanding of gravity when it was applied to larger scale, more complex, more distributed contexts such as astronomy
- ❑ Newton's clarification and formalization of the "force" of gravity overcame essentially all of these dissatisfactions
- ❑ But by Einstein's time, hardware technology (such as instrumentation range and precision) had raised a new set of incongruities between what was then understood as gravity and observable reality
- ❑ The understanding of gravity had to be generalized and elaborated by the law of relativity as "space-time curvature" in order to be better applicable in larger, more complex, more distributed contexts
- ❑ (Of course, now we know this remains a continuing process—c.f., "gravity waves")

Nature Provides Other Examples Of Paradigm Shifts To Accommodate Larger Scale

- ❑ Other examples of paradigm shifts for scaling up are readily found in nature,

  where higher animals are more complex because they are larger, rather than vice versa—e.g.,

  - * the principles of cell biology inherently limit the scale of single cell organisms
  - * the physiology of insects inherently limits their scale
- ❑ Examples are also manifest in engineering—e.g.,
  - * a small stream can be bridged by placing a log across it
  - * streams only a few times wider than the longest feasible single log can be bridged by joining a small number of logs end-to-end
  - * even wi.ler bodies of water require bridges based on entirely different principles, such as suspension

Asynchronous Decentralization Impacts The Nature Of Realtime Resource Management

- ❑ Asynchronous Decentralized Realtime Computing
- ❑ <u>A New Paradigm For Scaleable Realtime Computing</u>

We Propose A New Model Of Realtime Computing

- ❑ Because the traditional realtime viewpoint and its terminology is imprecise, oversimplified, and unrealistic,

  it can—an? does—limit

  - * the kinds of realtime systems that can be built
  - * the cost-effectiveness of those that are built
- ❑ Asynchronous decentralized realtime computer systems for mission management are a conspicuous instance of suffering from both these limitations
- ❑ We argue that our new paradigm of realtime computing offers a more systematic, comprehensive, and realistic framework which can help reduce such limitations—

  it is based on

  - * a new, more general method for expressing time constraints and scheduling objectives—

    the *Benefit Accrual Model*
  - * new realtime scheduling objectives and policies which accommodate the requirement for robust adaptivity to dynamic system and application conditions—

    Best-Effort policies

Our New Realtime Computing Paradigm Is Based On The Benefit Accrual Model And Best-Effort Scheduling

❑ The Benefit Accrual Model

❑ Best-Effort Scheduling

A Realtime Computation Has A Time Constraint

❑ We define a *realtime computation* to be a segment of a computational entity (such a thread, task, or process) subject to a time constraint

❑ A *time constraint* is the relationship between

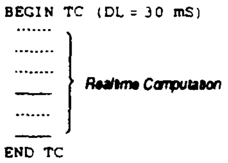
- ∗ when a realtime computation completes execution
- ∗ the *temporal merit* of that computation

e.g., in the classical deadline case

- ∗ completing before the deadline time is better
- ∗ completing after the deadline time is worse

❑ A time constraint is manifest in the computation program as a demarcated region of code whose execution completion time is subject tc the time constraint—

e.g., the computation must complete execution of the region before the deadline time arrives



otherwise it must suffer an exception condition

Timeliness Is The Basis For Realtime Scheduling

❑ We consider the *timeliness*—i.e., *temporal merit*—of computations to be the principle basis for

- ∗ specifying
- ∗ scheduling
- ∗ evaluating

computation completion times

❑ In the Benefit Accrual Model, timeliness is defined with a framework consisting of three relationships (e.g., functions)

A Timeliness Framework Is Comprised Of Three Parts

❑ Each realtime computation has a *time constraint*—i.e., a relationship between

- ∗ when the computation completes execution
- ∗ the resulting temporal merit—timeliness—of that computation

(e.g., for the classsical deadline time constraint, *lateness* = completion time − deadline)

❑ A *collective temporal merit* relationship defines

- ∗ the collective timeliness of a set of computations
- ∗ in terms of the individual timeliness of all its constituent computations

(e.g., the number of deadlines met—i.e., with negative lateness)

❑ A *collective temporal acceptability* relationship defines

- ∗ the *acceptability*—in an application-specific metric
- ∗ of the completion times—predicted or experienced—for a set of computations

  expressed in terms of their individual or collective timeliness
- ∗ for specified system and application states

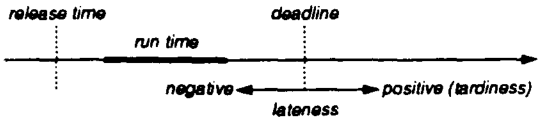
(e.g., acceptable means always meeting all deadlines)

Timeliness For Classical Deadline Time Constraints Is In Terms Of Tardiness

❑ The classical deadline time constraint (i.e., in scheduling theory) employs

- ∗ *lateness* = completion time − deadline
- ∗ or *tardiness* = positive lateness

as its individual measure of timeliness



❑ The collective timeliness relationship of a set of computations having classical deadline time constraints is most frequently chosen to be one of the following

- ∗ the occurrence or not of at least one tardy (positive lateness) completion
- ∗ the number of tardy completions
- ∗ the mean lateness

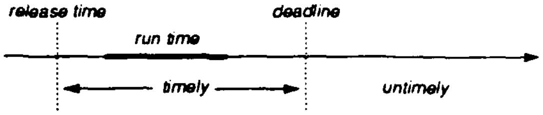
❑ Classical deadline-based scheduling theory often implicitly presumes that

collective temporal acceptability is equivalent to collective timeliness

The Traditional Hard Deadline Case Allows Only For Binary Timeliness And Acceptability

❑ The traditional realtime computing interpretation of "hard" deadlines implies restrictions of timeliness to

- ∗ a binary special case of the deadline time constraint—timely and untimely



- ∗ a binary collective timeliness relationship
  - ▪ untimely: the occurrence of at least one tardy completion
  - ▪ timely: otherwise
- ∗ a binary measure of collective temporal acceptability
  - ▪ acceptable: no occurrence of tardy completions (unanimous optimum) under any conditions
  - ▪ unacceptable: the occurrence of at least one tardy completion under any conditions

where the semantics of "unacceptable" are specific to the computation and application—e.g.,

- ▪ non-productive
- ▪ counter-productive

in some way

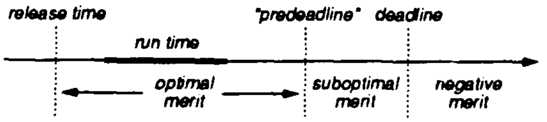
Often Time Constraints Are Not Binary

❑ Often it is very useful or necessary to have *softer*—i.e., non-binary—time constraints

❑ A common example of such a softer time constraint:

if a particular computation cannot be completed at a time of optimal merit—i.e., before its "predeadline"

- ∗ completing it a little "tardy" has reduced merit—but is better than not completing it at all
- ∗ however, completing it actually tardy (after its deadline) has negative merit—i.e., is worse than not completing it at all



❑ Some softer time constraints are routinely handled in terms of lateness with scheduling theory—

but the linearity of lateness greatly limits the interpretation of merit (e.g., excludes this example)

❑ Realtime computing practice tends to express and handle softer time constraints even less effectively—

not on a time constraint basis at all, but instead in disparate, ad hoc, imprecise ways

Often Collective Timeliness Is Not Binary

❑ Softer time constraints necessitate correspondingly "softer"—i.e., non-binary—collective timeliness relationships

❑ Using the previous time constraint example,

the collective timeliness relationship could be one which (as a scheduling criterion) increases the number of completions in the optimal region—e.g.,

- ∗ the sum (or mean) of
- ∗ weighted lateness = (completion time − deadline) + k (completion time − predeadline)

❑ Some softer collective timeliness relationships are routinely handled in terms of lateness with classical scheduling theory

while others necessitate more expressive time constraint relationships

❑ Realtime computing practice tends to express and handle softer collective timeliness less effectively—

not on a time constraint basis at all, but instead in disparate, ad hoc, imprecise ways

Often Temporal Acceptability Is Not Binary

❑ Softer collective timeliness necessitates correspondingly "softer"—i.e., non-binary—collective temporal acceptability relationships

❑ The degree of collective temporal acceptability might be based on

- ✶ collective timeliness alone—e.g., acceptable
  - only above one lower bound under certain circumstances, and above a different lower bound under other circumstances
  - to the degree that it exceeds a lower bound
- ✶ both individual and collective timeliness—e.g., acceptable to the degree that
  - some
    - ◊ total number of
    - ◊ or specific individual

    computations
  - are late by a certain amount
  - under certain conditions

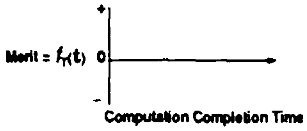
❑ Realtime computing practice tends to express and handle softer temporal acceptability less effectively—

not on a time constraint basis at all, but instead in disparate, ad hoc, imprecise ways

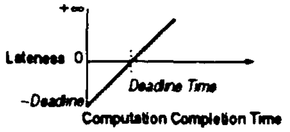
A Computation Time Constraint Relationship Is Temporal Merit As A Function Of Its Completion Time

❑ In the Benefit Accrual Model, a computation's time constraint relationship—i.e., urgency—is made arbitrary by thinking explicitly of

individual temporal merit being any function f_T of the computation's completion time t



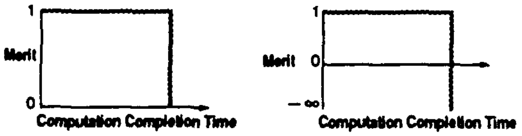
❑ The classical deadline function's merit of lateness is then depicted as



- ✶ a line
- ✶ with slope +1
- ✶ having a range of { − deadline, +∞}
- ✶ crossing the X axis at the deadline time (becoming tardiness)

The Traditional Realtime Computing Interpretation Of A Deadline Is A Downward Step Function

❑ The traditional realtime computing interpretation of a deadline, when viewed as a time constraint function, is



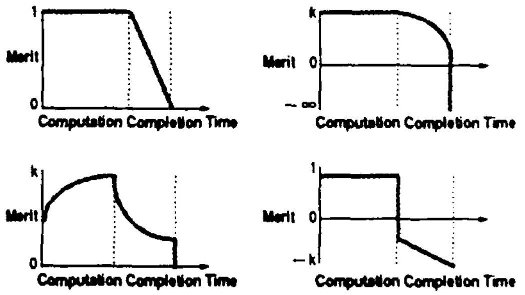
- ✶ a binary-valued, downward step function
  - completing the computation anytime between its release (X = 0) and deadline times is uniformly timely
  - and otherwise is uniformly untimely
- ✶ the smaller of the two binary merit values may be
  - 0: zero merit is attained for completing the computation after its deadline
  - −∞: a large merit penalty is incurred for completing the computation after its deadline

In Real Systems Very Often The Time Constraint Is Neither Linear Nor Binary

❑ Both the classical and traditional realtime computing interpretations of a deadline are often poor approximations to actual realtime constraints

❑ There are many cases in realtime applications where

- ✶ there is some diminished merit attained for completing the computation within an allowable tardiness period
- ✶ the merit is not constant prior to the "deadline"
- ✶ the penalty is not constant after the "deadline"
- ✶ the merit measure and range are application-specific

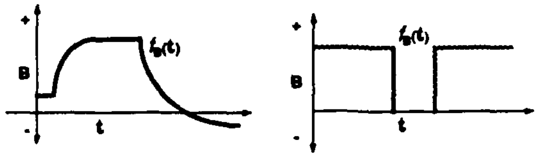


❑ Deadlines are not a general mechanism for expressing scaleable realtime time constraints

In The Benefit Accrual Model A Time Constraint Is Expressed By A Benefit Function

❑ The Benefit Accrual Model expresses an individual computation's time constraint relationship in terms of a temporal merit called *benefit* (B)

❑ Benefit functions may be *unimodal* or *multimodal*



(the non-linear optimizations involved in dealing with multimodal benefit functions lead us to temporarily confining ourselves here to unimodal ones)

❑ The benefit metric is application-specific and defined system-wide

❑ Benefit functions are

- ∗ derived by the programmers directly from the requirements and behavior of the realtime computation (usually an application activity)
- ∗ subject to a system-wide engineering process (just as are assignments of classical priorities)

A Benefit Function Is Defined Over A Range Of Time

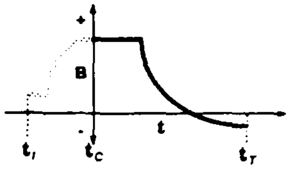
❑ The time axis is the one the scheduler uses—it may be

- ∗ physical
  - ▪ absolute ("calendar/wall clock") time
  - ▪ relative to (since) some past event
- ∗ logical—a number which monotonically increases, but not necessarily at regular intervals

❑ The origin of the benefit function axes is the current time t_C (value of the system clock)

❑ The earliest time for which a benefit function is defined is called its *initial time* t_I;

the latest time for which a benefit function is defined is called its *terminal* time t_T



(some systems and scheduling algorithms call for the specification of an indefinite terminal time)

❑ A benefit function is evaluated only for values of its time parameter between the current time and its terminal time

Sooner And Later Times Define The "Best" Interval

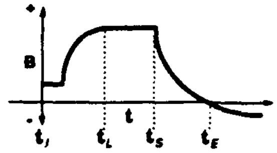
❑ The *later* time t_L is that after which the benefit function value is (monotonically) non-increasing

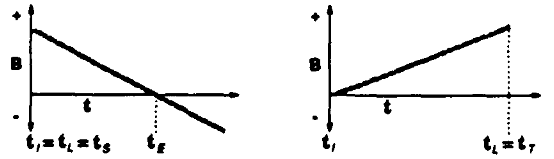
- ∗ thus, completing the realtime computation at or after this time is better
- ∗ a benefit function always has a t_L

❑ The *sooner* time t_S is that after which the benefit function value is (monotonically) decreasing

- ∗ thus, completing the realtime computation at or before this time is better
- ∗ a benefit function need not have a $t_S < t_T$

❑ If its value becomes zero or negative at time $t_E \geq t_S$, a benefit function has an *expiration* time





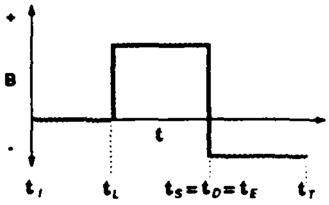
Deadlines Are Due Times Subject To A Specific Collective Temporal Acceptability Criterion

❑ A special case of a sooner time t_S is a *due* time t_D, distinguished by the benefit function's first derivative having an infinite discontinuity at $t_S = t_E$

❑ A *deadline* is a due time subject to a collective temporal acceptability criterion which does not allow the due time to be missed

❑ A benefit function is defined as *hard* if it has

- ∗ a zero or constant negative value before t_L
- ∗ an infinite discontinuity in its first derivative at t_L if $t_L > t_I$
- ∗ a due time t_D
- ∗ a constant value between t_L and t_D
- ∗ a constant value between t_D and t_T



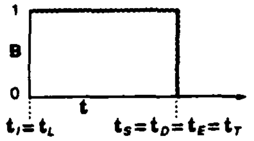
A Classical "Hard Deadline" Is A Special Case

- ❑ The most common meaning of a classical "hard deadline"—

  a computation which completes anytime between its initial and deadline times is uniformly acceptable, and otherwise is unacceptably tardy—

  corresponds in this model to

  - ✱ a hard benefit function with
  - ✱ deadline $t_D = t_T$
  - ✱ unit binary range {0, 1}

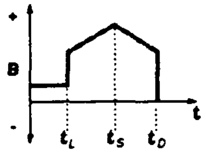


- ❑ Classical definitions of "hard deadline" vary a little
  - ✱ they generally do not provide for a $t_L > t_I$
  - ✱ sometimes the range of this function is $\{-\infty, 1\}$; a few algorithms define the range as $\{0, ke\}$,

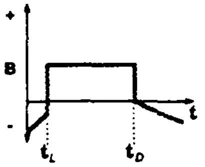
    where e is the computation's execution time and k is a proportionality factor

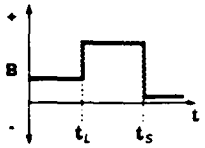
All Benefit Functions Which Are Not Hard Are Soft

- ❑ All benefit functions which are not hard are *soft*
- ❑ Soft benefit functions can have arbitrary values before and after the optimal value at t_S



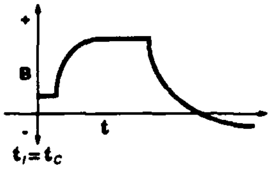
- ❑ Soft benefit functions need not have
  - ✱ constant values on each side of t_L and t_D
  - ✱ expiration times



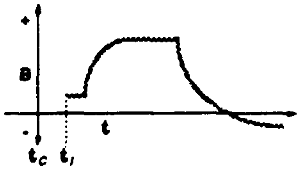


A Released Time Constraint May Be Effective Either Immediately Or In The Future

- ❑ A time constraint—and thus benefit function—is made known to the scheduler at its *release* time (which is usually a scheduling event)
- ❑ When the benefit function is released, its initial time may be
  - ✱ the current time—the time constraint is released at the time it is to take effect (i.e., at $t_I = t_C$)



  - ✱ a future time—the time constraint is released in advance (i.e., $t_I > t_C$) to improve scheduling



(but $t_I \le t_C$ is a necessary condition for the computation to complete, if not also begin, execution)

Realtime Computations Generally Have Dynamic Dependencies

- ❑ Expressing or releasing a benefit function relative to a future time/event, such as
  - ✱ the completion of some other computation
  - ✱ an external signal

  is adding a (generally dynamic) dependency to the time constraint

- ❑ Dynamic dependencies can require a realtime computation to be completed at a time yielding zero or negative benefit, when a computation
  - ✱ has been initiated and cannot be
    - ▪ stopped (preempted or aborted)
    - ▪ undone

    (such as one related to a physical activity in the application environment)
  - ✱ would block another if not completed, despite its consequential zero or negative benefit

  (which can require indefinite function terminal times)

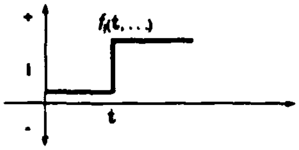
- ❑ Dependencies must be accommodated in conjunction with time constraints according to some specific scheduling policy,

  and thus are not part of the benefit accrual model per se

Computations Also Have Relative Importances

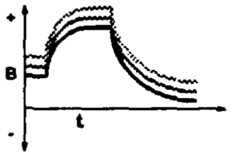
- ❑ Each computation generally also has a relative *importance*—i.e., functional criticality—with respect to other computations contending for completion
- ❑ Importance is orthogonal to urgency
  - ∗ a computation with high urgency (e.g., a near deadline) may not be highly important
  - ∗ a computation with low urgency (e.g., a far deadline) may be very important
- ❑ Importance may be a function f_I of time and other parameters that reflect the application and computing system state,

  and can be represented and employed similar to urgency



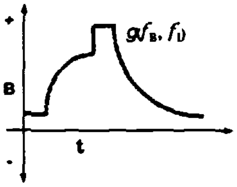
Urgency And Importance Are Used Together

- ❑ In simple cases importance may be a constant, and benefit may be simply urgency scaled by importance
  - ∗ urgency and importance might be combined prior to execution time
  - ∗ a computation's completion might be expedited by elevating its benefit for the remaining execution time



- ❑ In more general cases where importance needs to be a variable, f_B and f_I must be evaluated together dynamically to determine the benefit—

  e.g., as some function of the f_B and f_I functions, $g(f_B, f_I)$



Realtime Schedulers Are Usually Presumed To Know Something About Computation Execution Durations

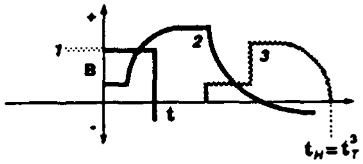
- ❑ A realtime computation has an execution duration *e* which the scheduler
  - ∗ either knows prior to execution—important for realtime scheduling
    - ▪ deterministically (the most common presumption)
    - ▪ estimated
      - ◊ stochastically (i.e., in expectation)
      - ◊ non-stochastically—e.g.,
        - ~ bounds
        - ~ rules
  - ∗ or does not know prior to execution—limits predictability of realtime scheduling
- ❑ This duration may or may not take into account a forecast of dynamic dependencies
- ❑ Non-deterministic durations may be estimated dynamically (during the computation's execution)—e.g.,
  - ∗ conditional probability distributions
  - ∗ execution-time knowledge-driven rules

The Benefit Accrual Model Is Based On Benefit Functions And Benefit Accrual Functions

- ❑ Benefit Functions
- ❑ <u>Benefit Accrual Functions</u>

The Scheduler Assigns Execution Completion Times

□ A benefit accrual model scheduler considers all released time constraints between the current time and its *horizon* t_H—the future-most terminal time



□ It assigns the estimated execution completion times—and consequently the

- ∗ initiation times
- ∗ ordering

for those computations

using an algorithm which

- ∗ seeks to sufficiently satisfy the scheduling—collective temporal acceptability—criterion
- ∗ taking into account dependencies and importances

(such as *earliest-deadline-first* for the classical "hard realtime" criterion of all computations meeting their deadlines)

Collective Optimality Is Not Always Defined In Terms Of Unanimous Individual Optimums

□ For the special case of any collective temporal acceptability criterion defined to be a

- ∗ unanimous
- ∗ optimum

of the individual temporal acceptabilities,

there is an equivalent criterion defined in terms of individual, rather than collective, optimums—e.g.,

- ∗ meet all deadlines—meet each deadline
- ∗ maximize all benefits—maximize each benefit

□ In general, collective temporal acceptability is not defined as necessarily unanimous or optimum with respect to the individual computations' temporal acceptability—e.g., maximize

- ∗ the number of deadlines met
- ∗ the sum of the benefits
- ∗ the number of computations during a time frame T which achieve at least P percent of their maximum possible benefit
- ∗ the probability that at least P percent of the computations during a time frame T will achieve their maximum benefits

In The Benefit Accrual Model Collective Temporal Acceptability Is Based On Accruing Benefit From Individual Computations

□ In the benefit accrual model, collective temporal acceptability criteria are based on

- ∗ *accruing* benefit from the individual computations in a set
- ∗ in a manner specified by a *benefit accrual function* for that set

□ This is general enough to encompass a wide range of collective temporal acceptability criteria

- ∗ the unanimous individual optimum cases such as traditional "hard realtime,"

  for which the accrual predicate is the product of the individual benefits (assuming the usual range of {0, 1})
- ∗ cases not defined as necessarily unanimous or optimum with respect to the individual computations' temporal acceptability,

  which we term *best-effort* scheduling

Our New Realtime Computing Paradigm Is Based On The Benefit Accrual Model And Best-Effort Scheduling

□ The Benefit Accrual Model

□ Best-Effort Scheduling

digital

Conventional Realtime Scheduling Focuses On Unanimous Optimum As The Criterion

- ❑ Scheduling principles and practices which are realtime by our definition (i.e., based on satisfying completion time constraints) have until recently been focused exclusively on
  - ✱ guaranteeing that
  - ✱ a unanimous optimum

  scheduling criterion will be met

  (e.g., the classical "hard realtime" case of guaranteeing that all deadlines are always met)

- ❑ Even though the traditional "hard realtime" cases are intended—and commonly imagined—to achieve this ideal
  - ✱ physical laws (especially in decentralized systems)
  - ✱ or the intrinsic nature of the applications (especially at mission management levels)

  generally make it
  - ✱ either non-cost-effective
  - ✱ or impossible

  (there are only a few exceptions)

digital

Realtime Computing Systems Generally Have A Wide Spectrum Of Mission-Critical Timeliness Needs

- ❑ In general, realtime systems need
  - ✱ a sufficient number of computation completion times to be
  - ✱ sufficiently likely
  - ✱ to be sufficiently acceptable (perhaps optimal)
  - ✱ given the current application and computer system circumstances
  - ✱ (perhaps over a wide range of such circumstances)

  where each instance of "sufficient" is application-specific

- ❑ The Benefit Accrual Model provides a framework for expressing
  - ✱ "softer"
    - ▪ time constraints—in the sense of non-binary completion time acceptability
    - ▪ scheduling criteria—in the sense of non-unanimous and non-optimum
  - ✱ in addition to—and in the same manner as—the conventional "hard" time constraints and scheduling criteria

- ❑ These softer needs are realized with *best-effort* scheduling algorithms

digital

Best-Effort Scheduling Seeks To Do The Best That Is Possible Under The Current Conditions

- ❑ "Best-effort" (BE) realtime scheduling algorithms seek to provide the "best"—as specified by the application— computational timeliness they can,

  given the current application and computer resource conditions

- ❑ This concept,

  and the Time-Value Function progenitor of the Benefit Accrual Model as a framework for expressing time constraints,

  were originated by Jensen in 1977 and published in 1985

- ❑ The first generation of BE—on-line (at execution time)—scheduling algorithms emerged from Jensen's Ph.D. students in his Archons Project at CMU, for the Alpha asynchronous decentralized realtime OS kernel
  - ✱ Locke's algorithm (1986)
  - ✱ Clark's algorithm (1990)

- ❑ A second generation of on-line BE algorithms is being devised as part of a recent multi-university effort to establish formal performance bounds for on-line algorithms in general and certain BE ones in particular

- ❑ A first generation of off-line BE algorithms is being devised in France

digital

Locke Did The First Best-Effort Scheduling Algorithm

- ❑ The most salient characteristics of Locke's algorithm
  - ✱ allows a wide variety—but not all forms—of Time-Value Functions (TVF's)
  - ✱ intends that importance be reflected by scaling the TVF values
  - ✱ execution times are defined stochastically
  - ✱ when underloaded, schedules Earliest-Deadline-First (EDF) to meet all deadlines—which in generai does not accrue maximum value
  - ✱ if a job arrival, or execution time overrun, results in a sufficiently high probability of overload,

    jobs are set aside in order of minimum *expected value density* (expected value/expected remaining execution time) until the probable overload is removed
  - ✱ the scheduling optimality criterion when overloaded is the special (but reasonable) case of maximizing the sum of the job values attained
  - ✱ does not deal with dependencies (e.g., precedence, resource conflicts)

- ❑ Locke used simulations to demonstrate that his algorithm performed well in comparison to others, such as EDF, for a number of interesting overload cases;

  but provided no formal performance characterizations

Locke's Algorithm Has Been Used Experimentally

❑ Versions of Locke's algorithm have been implemented and experimentally verified to be superior and cost-effective

with respect to traditional realtime scheduling algorithms, such as EDF and fixed priority,

for a number of interesting cases—including

- ∗ in the Alpha asynchronous decentralized realtime OS kernel
  - ▪ a battle management application for air defense, by General Dynamics and the Archons Project at CMU, in 1987
  - ▪ a ball-and-paddle realtime scheduling evaluation testbed by the Archons Project in 1987

  which also added

  - ▪ nested time constraints
  - ▪ timeliness failure abort processing
- ∗ in the Mach 2.5 OS kernel
  - ▪ a synthesized realtime workload, by the Archons Project in 1987

Clark's Algorithm Deals With Dependencies

❑ The most salient characteristics of Clark's algorithm

- ∗ permits only rectangular TVF's, whose value is the job's importance
- ∗ execution times are both fixed and known
- ∗ the scheduling optimality criterion is the special (but reasonable) case of maximizing the sum of the job values attained
- ∗ deals with dependencies (e.g., precedence, resource conflicts) which are not known in advance
- ∗ selects jobs to be scheduled in decreasing order of value density (VD)
- ∗ selected jobs are scheduled EDF, which maximizes summed value for the TVF's he permits
- ∗ when each job is scheduled, so are those on which it depends
- ∗ if necessary, precedent jobs are aborted or their deadlines are shortened (whichever is faster), to satisfy the deadline of the dependent job

❑ Clark's formal analysis and simulations showed that

- ∗ when overloaded, if the algorithm can apply all available cycles to jobs that complete, no other algorithm can accrue greater value given the current knowledge
- ∗ since future jobs are unknown, there is no performance guarantee

Recent Work Explores Competitive Factor Bounds

❑ Researchers at UTexas, NYU, and UMass have recently developed limited performance bounds for on-line realtime scheduling

- ∗ *competitive factor* measures the value an algorithm guarantees it will achieve compared to a clairvoyant scheduler
- ∗ considers only rectangular TVF's, and execution times which are (mostly) both fixed and known—

  like Clark's algorithm, this means that scheduling by EDF when underloaded not only meets all deadlines but maximizes summed value
- ∗ if all values are proportional to execution time, an on-line algorithm can guarantee a competitive factor of no more than 1/4
- ∗ the performance bound is lower when
  - ▪ value is not proportional to execution time
  - ▪ the ratio of maximum to minimum VD increases
  - ▪ execution times are not fixed and known
- ∗ confirms that performance guarantees are impossible if workload characteristics are unknown
- ∗ acceptable performance assurances may be possible when limited, reasonable, workload information is known

❑ Their algorithms are devised primarily for the purpose of illustrating the performance bound

Maynard Is Addressing Overload Behavior With Best-Effort Schedulers

❑ Maynard's thesis is

- ∗ improving the understanding of the overload behavior of on-line realtime scheduling algorithms
- ∗ developing techniques for defining benefit functions to yield desired overload behavior

❑ Its scope includes best-effort schedulers that use benefit density as the load shedding criterion

❑ The work to date provides an algorithm for setting job importance values to impose a strict priority ordering among selected groups of jobs

❑ This allows integration of results from off-line schedulability analysis, to

- ∗ provide "guarantees" when necessary and possible
- ∗ retain adaptability of dynamic scheduling

❑ His simulations support the validity of the approach

❑ He is also creating tools which help the system designer

- ∗ select and adapt suitable scheduling algorithms for specific applications
- ∗ choose appropriate job importance values

There is Somewhat Related Work In Other Fields

- ❑ The most closely related work to Best-Effort realtime scheduling is Cost-Based Scheduling for queueing and dropping network packets, done at Stanford
  - ∗ a cost function specifies the cost per unit length of queuing delay for a packet as a function of time
  - ∗ ,ackets have only non-decreasing cost functions
  - ∗ instead of creating a schedule, the algorithm queues the next packet which it estimates would cost the most to delay
  - ∗ cost is calculated using a estimation of future cost that would be incurred, which is the same for all packets
  - ∗ the optimization objective is to minimize the average delay cost incurred by all packets
  - ∗ dependencies are not considered
  - ∗ their simulations show that the algorithm performs well compared to the standard packet queuing algorithms, and Locke's algorithm,

    for certain workloads—packets averaging unit length, in near fully loaded conditions
- ❑ These premises do not correspond well to workload characteristics of general interest in realtime computation job scheduling

Best-Effort Benefit Accrual Scheduling Exacts A Higher Price Than Simpler Approaches

- ❑ Benefit functions and best-effort realtime scheduling algorithms
  - ∗ utilize more application-supplied information than is usual
  - ∗ place specific requirements on the kind of scheduling mechanisms that must be provided (i.e., in the OS kernel)
  - ∗ and thus exact a higher computational price than when little or no such information is used
- ❑ In many (if not most) cases, high cost/performance can be attained by good engineering
- ❑ Much of the price can be paid with inexpensive hardware
  - ∗ higher performance processors
  - ∗ a dynamically assigned processor in a multiprocessor node
  - ∗ a special-purpose hardware accclerator (analogous to a floating-point co-processor) in a uniprocessor or multiprocessor node

Best-EffortBenefitAccrualSchedulingWillBeSupported In A New Version Of The OSF Mach 3 Standa.d

- ❑ Version 5.0 of the Mach 3 microkernel standard from OSF has no realtime capabilities
- ❑ To create realtime functionality for subsequent versions of the OSF Mach 3 microkernel standard,

  a team of organiza.ions is collaborating—primarily
  - ∗ Digital Equipment Corp.'s Libra program
  - ∗ OSF's Research Institute
  - ∗ WPI's Center for High Performance Computing
  - ∗ SRI International

  with additional funding from
  - ∗ DARPA
  - ∗ USAF Rome Labs
  - ∗ Digital early-adopter customers
- ❑ This new realtime functionality will include
  - ∗ kernel mechanisms to implement virtually any scheduling policy specified by the client—

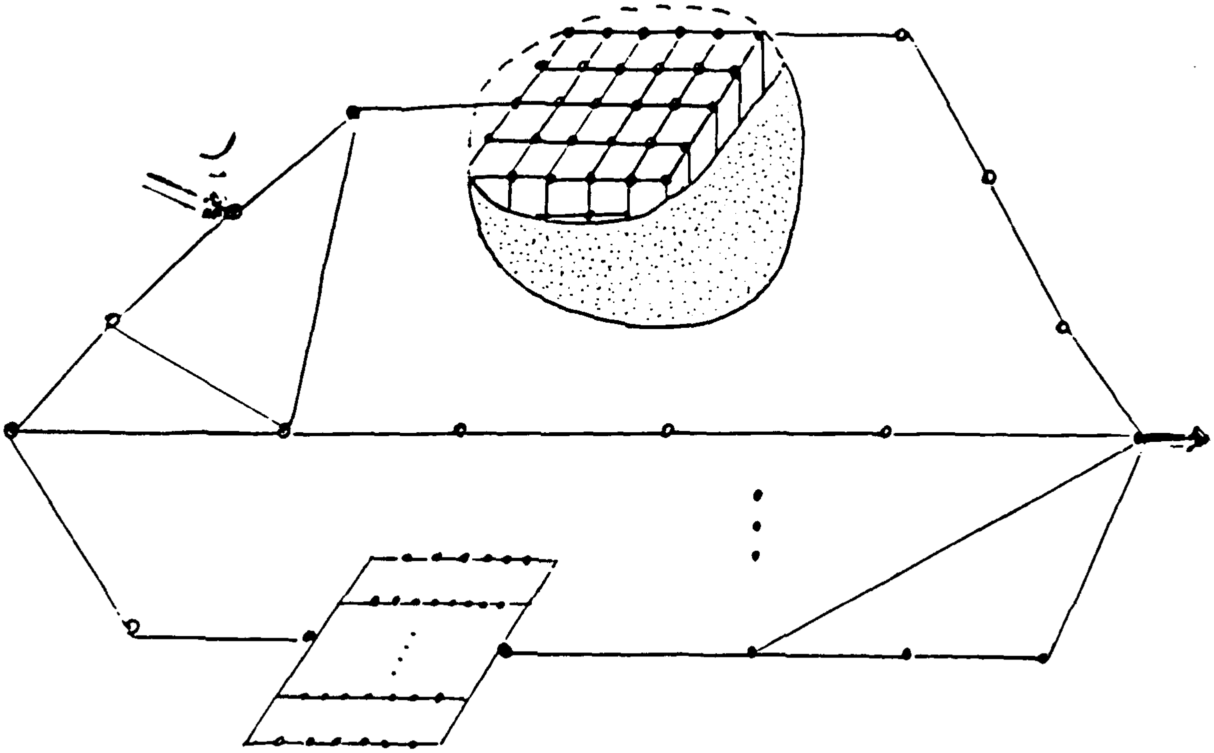
    specifically includiug best-effort benefit accrual ones
  - ∗ a scheduling policy interface to the kernel mechanisms that facilitates the creation, maintenance, and replacement of policies

Realtime Trans-Node (Alpha Kernel) Threads Are Also Being Incorporated Into The Mach 3 Standard

- ❑ The same team is also incorporating the Alpha kernel's realtime trans-node threads into forthcoming versions of OSF's Mach 3 standard—

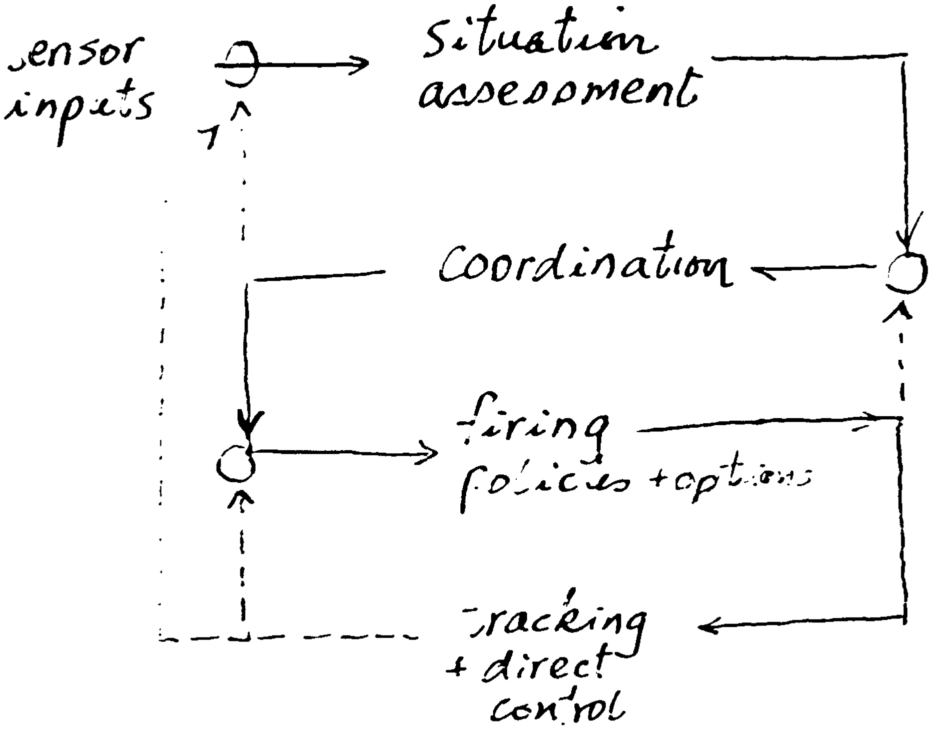
  this will greatly improve the ability to construct asynchronous decentralized systems (among other things)

Distributed and Parallel Environments



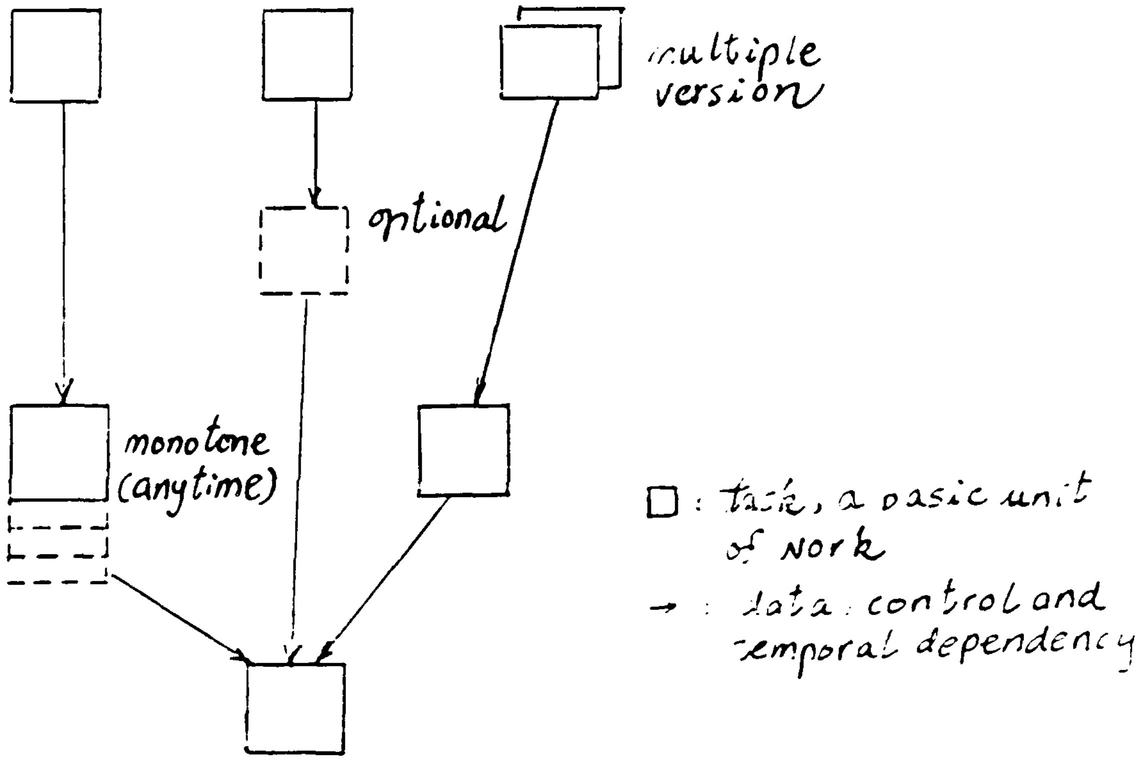
How to build real-time systems in such environments that can be systematically validated, tested and certified?

- Reference model(s) of real-time systems – where and what are the interfaces!
- Composition theory – how to construct large systems from building blocks and reason about the timing behavior of the composite system
- Environments containing building blocks and tools to support the decomposition and composition of systems and validate and evaluate their real-time performance
- Methods to ensure graceful degradation
- Theories to support validation and testing
- Methods and tools for measurements and experimentation
- Algorithms for end-to-end scheduling and resource management



control hierarchy

Imprecise computations: a way to provide graceful degradation

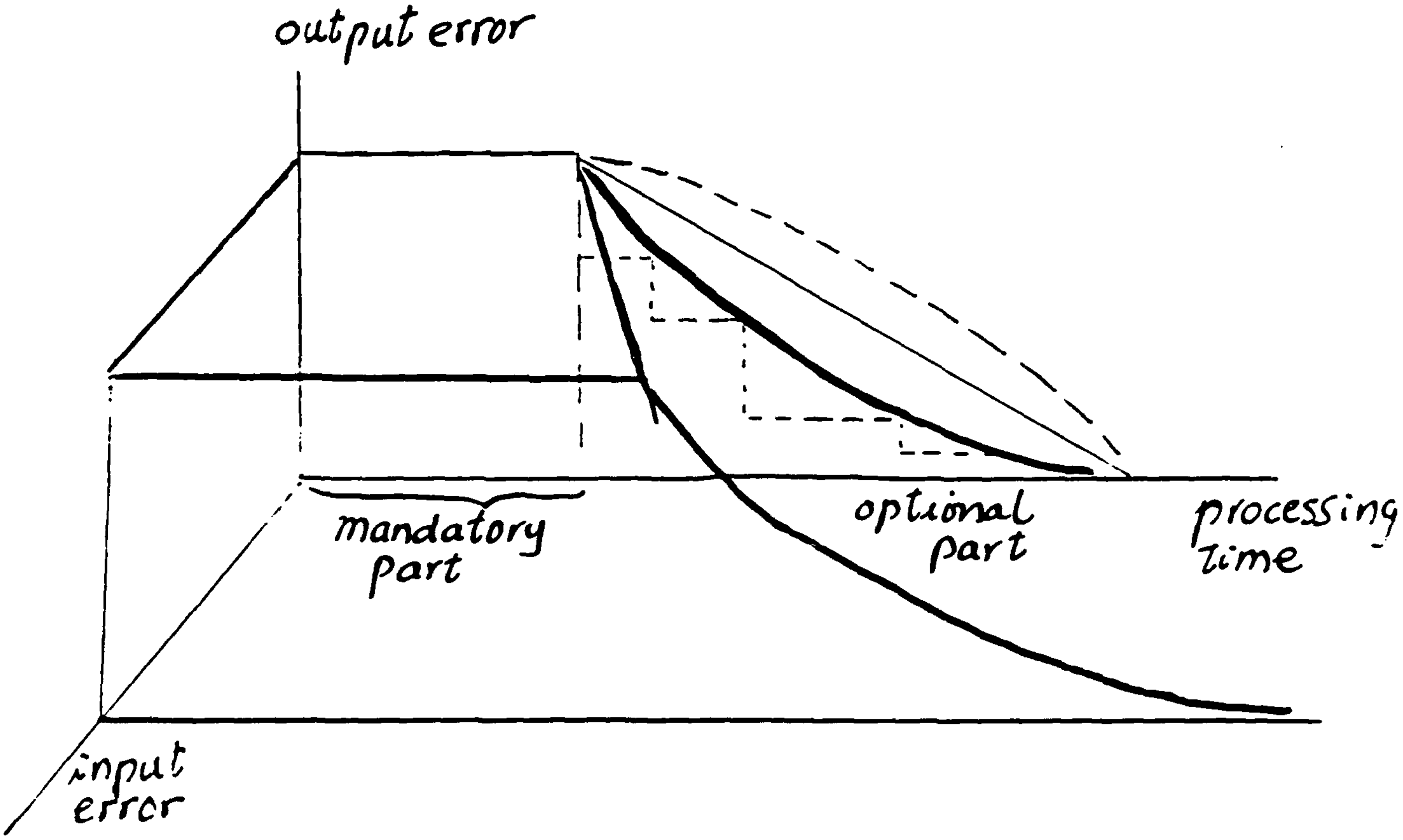

multiple
version
optional
monotone
(anytime)
□ : task, a basic unit of work
→ : data, control and temporal dependency

We have algorithms for scheduling, on-line & off-line

- monotone, periodic and complex tasks
- optional tasks with 0/1 constraint and multiple versions

and are working one algorithms for

- end-to-end scheduling
- scheduling tasks with input errors

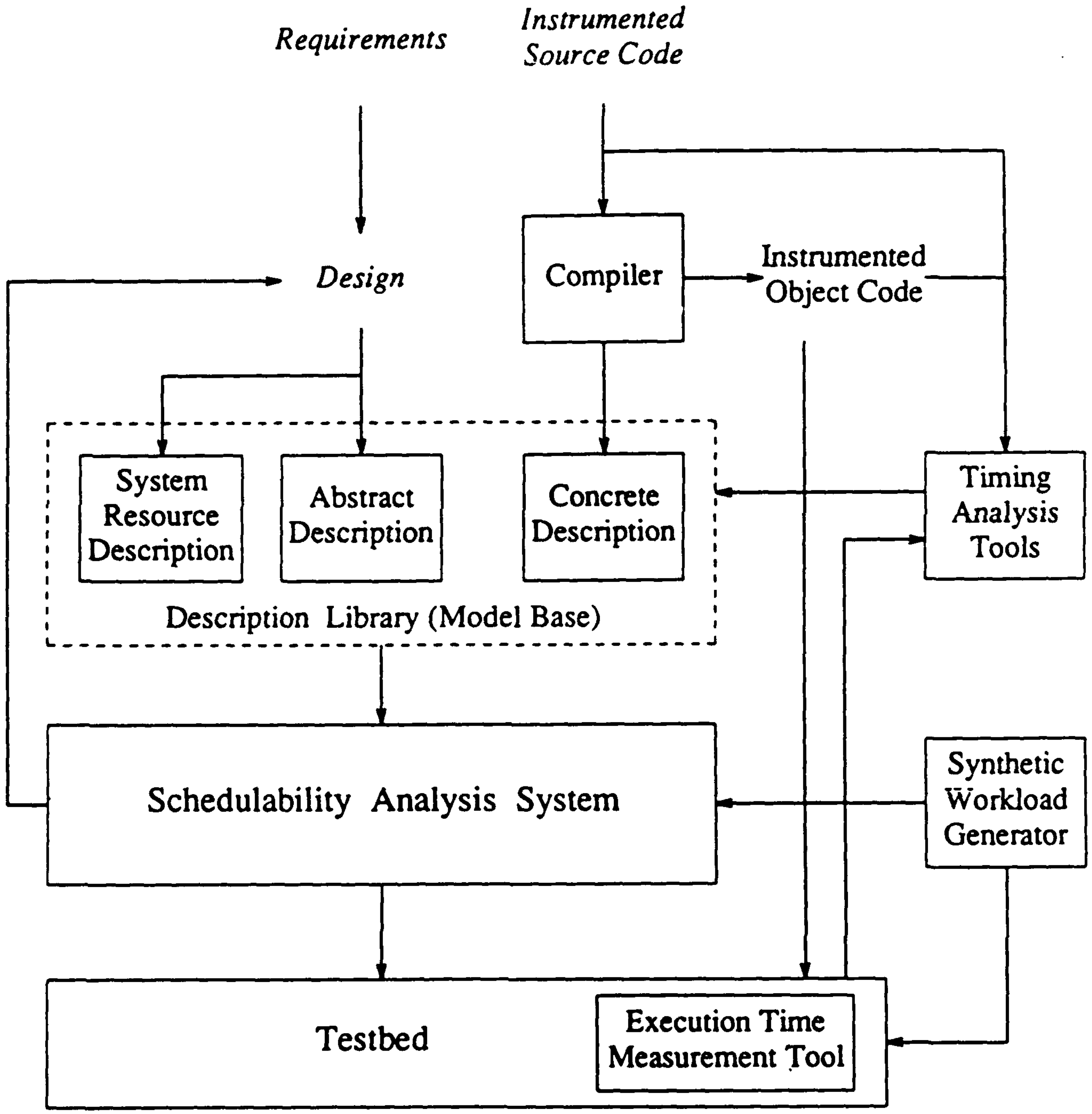


Quality Vs time

A system has a range of requirements:

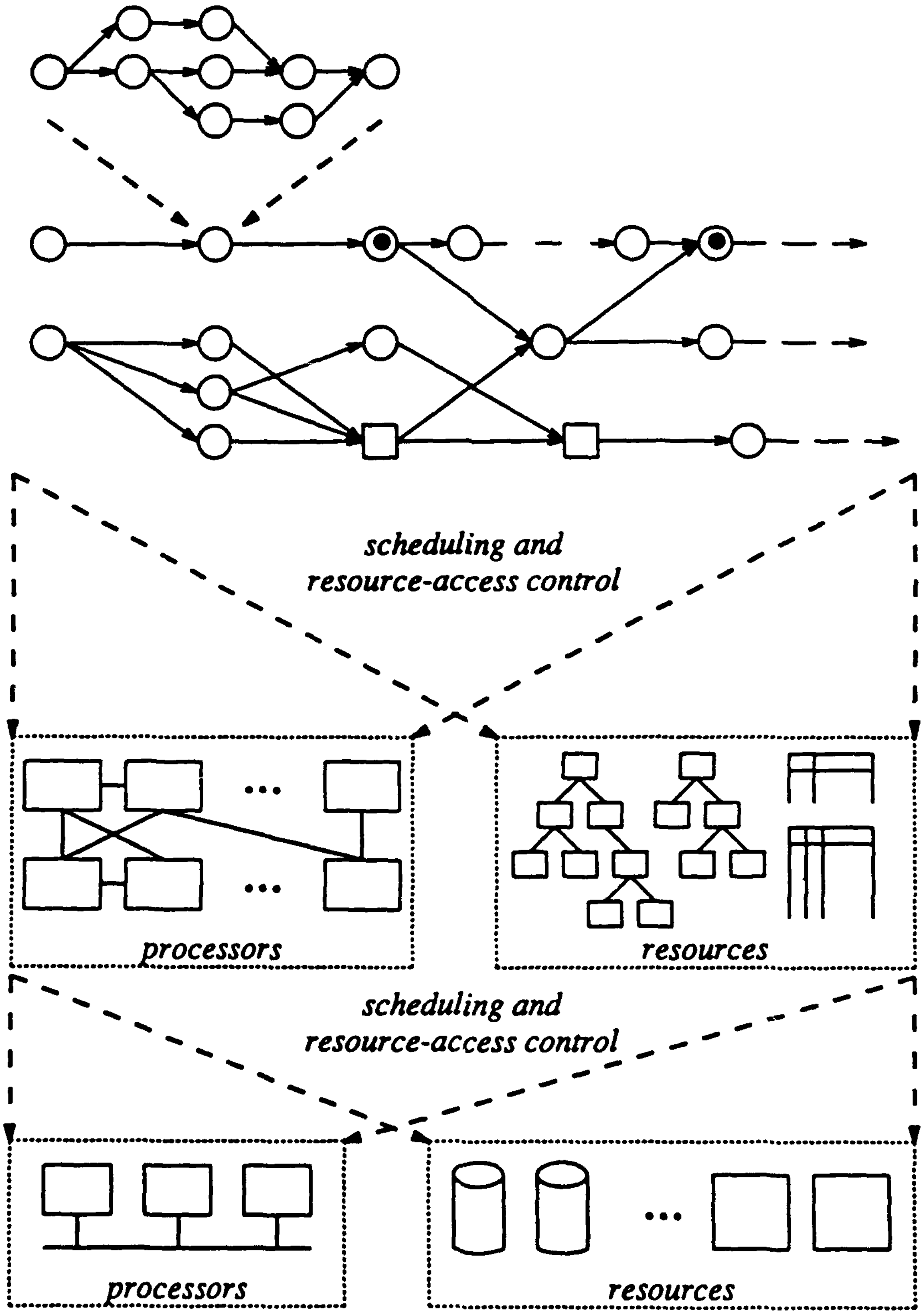
minimal (acceptable) requirements → nominal requirements

PERTS

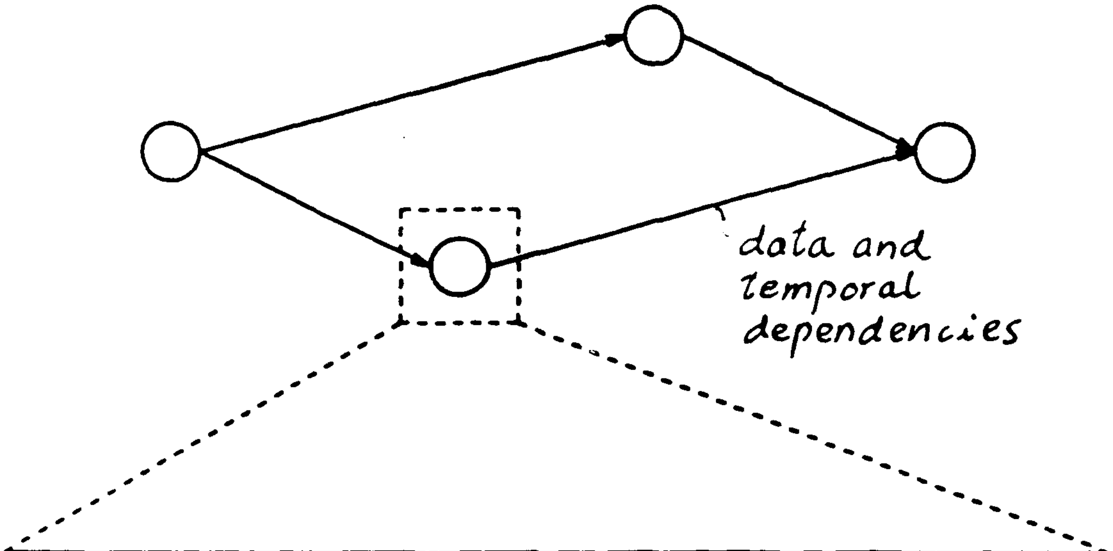

Requirements
Instrumented Source Code
Design
Compiler
Instrumented Object Code
System Resource Description
Abstract Description
Concrete Description
Description Library (Model Base)
Timing Analysis Tools
Schedulability Analysis System
Synthetic Workload Generator
Testbed
Execution Time Measurement Tool

- contains a collection of schedulers and resource managers, together with tools for validation and evaluation of the system using them

- As an *interactive design tool*
  - The user provides
    - task system description (an annotated data-flow graph of the task system),
    - system resource description, and
    - information on additional external constraints.
  - Outputs produced by PERTS include
    - processor and resource requirements,
    - sample task partitions and allocations,
    - sample schedules and memory layouts,
    - performance predictions, and
    - suggested design changes and tests.
- As a *development and evaluation tool*
  - The user provides
    - annotated source code or object interface definitions, and
    - system description.
  - PERTS can provide
    - a simulated (or emulated) target environment, and
    - performance profile.


scheduling and
resource-access control
processors
resources
scheduling and
resource-access control
processors
resources

Task System Description



— *Ready Time* (0)

— *Deadline* (infinity)

— *Period* (infinity)

— *Phase* (0)

— *List of resource requirements*
(types, units and required intervals)

— *List of optional intervals* (null)

— *In type* (AND)

— *Out type* (AND)

— *Laxity type* (better-late-than-never)

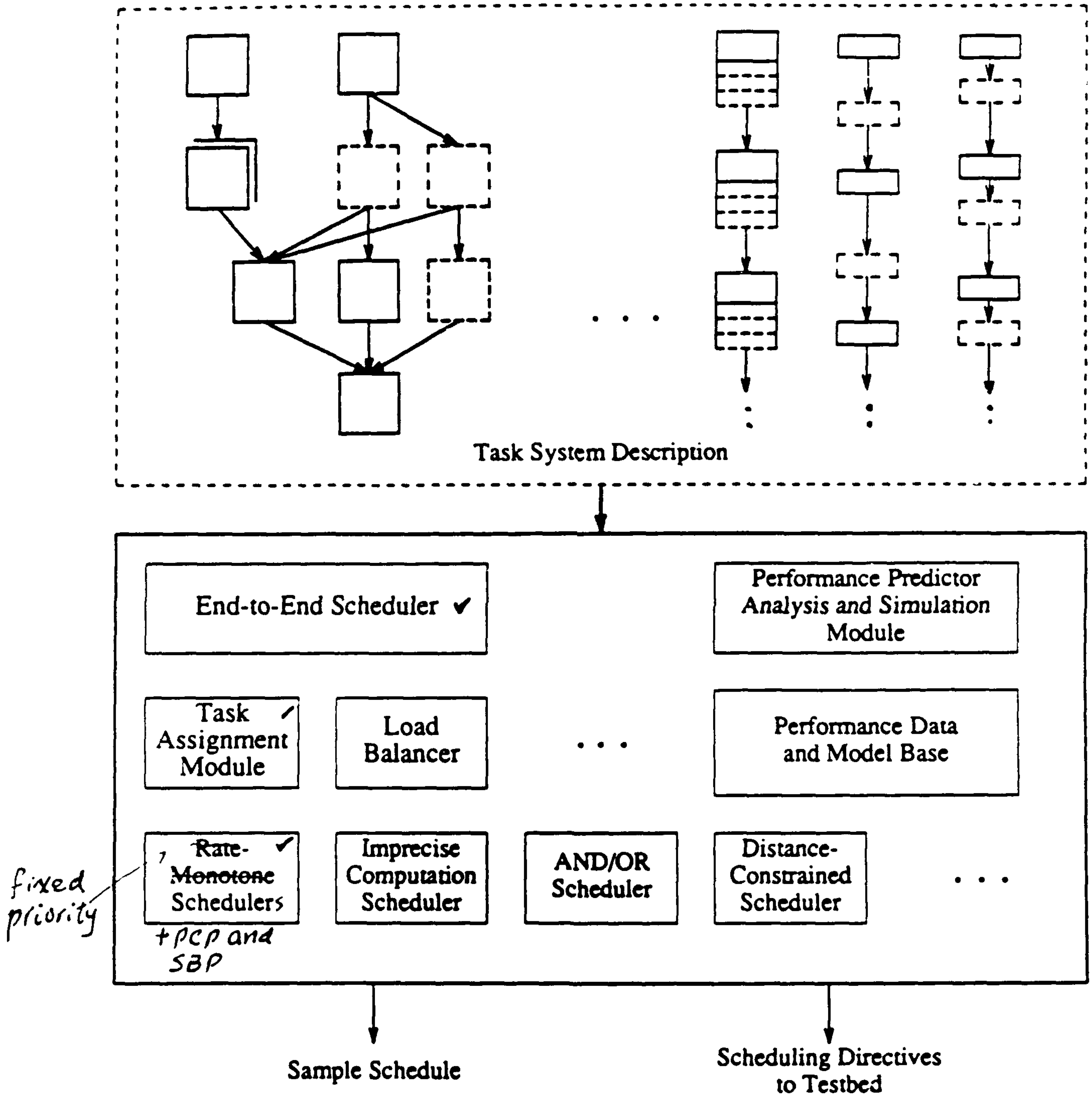
attributes of a task: a unit of work

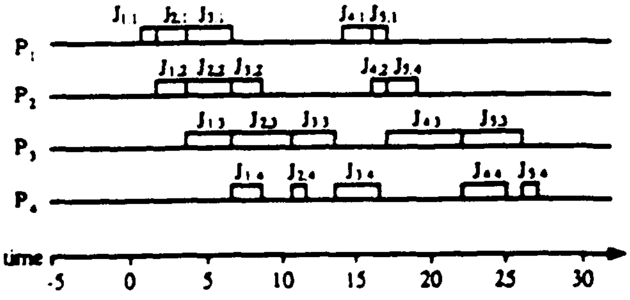
Other Input Information

- **System description** — a list of resources, each defined by parameters including
  - — *acquisition time* (time required to acquire an idle resource)
  - — *de-acquisition time* (time required to release a resource)
  - — *latency time*
  - — *context switch time* (time to switch the resource from one task to another if preempted)
  - — *preemptability* (whether the resource must be used serially)
  - — *maximum number of owners*
  - — *number of current owners*

- **External Constraints** — Arbitrary constraints that cannot be deduced from task and resource descriptions. Examples are maximum allowable processor utilization, intentional idle resources, nonpreemptive tasks, etc.

Schedulability Analysis System


Task System Description
End-to-End Scheduler ✓
Performance Predictor Analysis and Simulation Module
Task Assignment Module ✓
Load Balancer
. . .
Performance Data and Model Base
~~Rate-Monotone~~ Schedulers ✓
fixed priority
+PCP and SBP
Imprecise Computation Scheduler
AND/OR Scheduler
Distance-Constrained Scheduler
. . .
Sample Schedule
Scheduling Directives to Testbed



- Variations in processing time and resources required by individual tasks due to
  - — data-dependent execution
  - — effects of performance enhancing features
  - — resolution and error in processor time and resource usage measurements, etc.
- Variations in dispatching and execution orders when
  - — tasks content for resources
  - — there are data and control dependencies
  - — ready times of tasks are arbitrary
- Variations in the number of tasks

ARE UNAVOIDABLE

- A *priority-driven* or *list* scheduling algorithm

  — assigns priorities to tasks,

  — makes scheduling decisions and, possibly, alters task priorities

  ∘ when any task becomes ready and

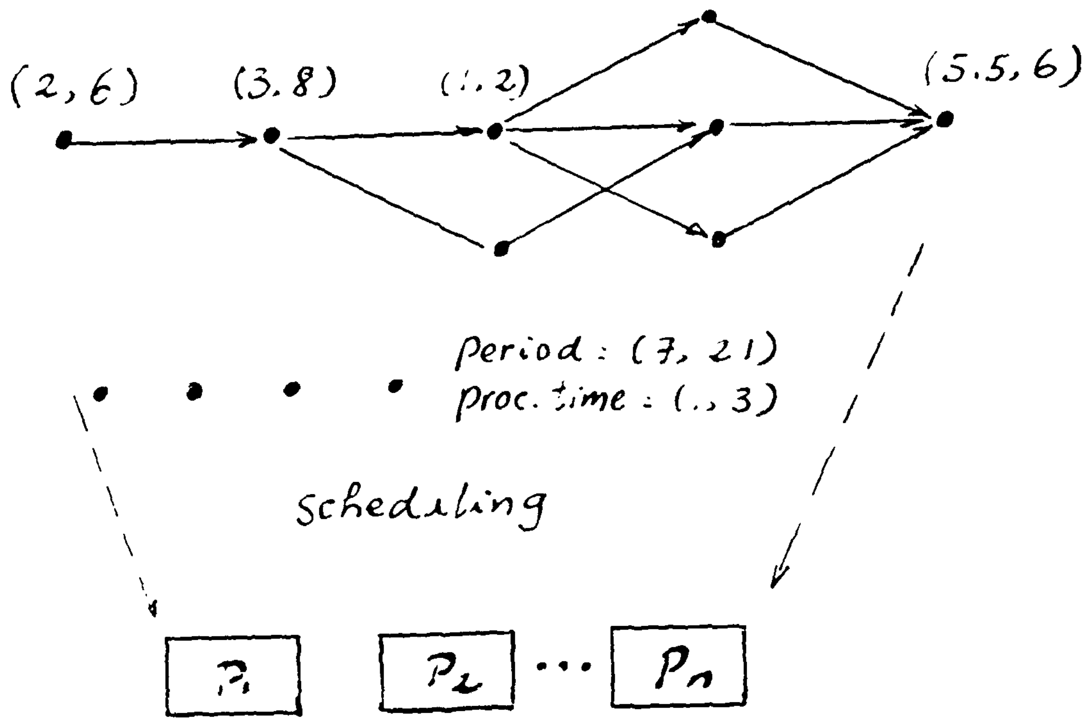
  ∘ when any task completes, and

  — at each scheduling decision time, executes the task with the highest priority among all ready tasks.

- All algorithms that never leave the processor(s) idle intentionally are priority-driven algorithms.

- Examples are rate-monotonic, earlist-deadline-first, shortest-processing-time-first and first-in-first-out.

Given



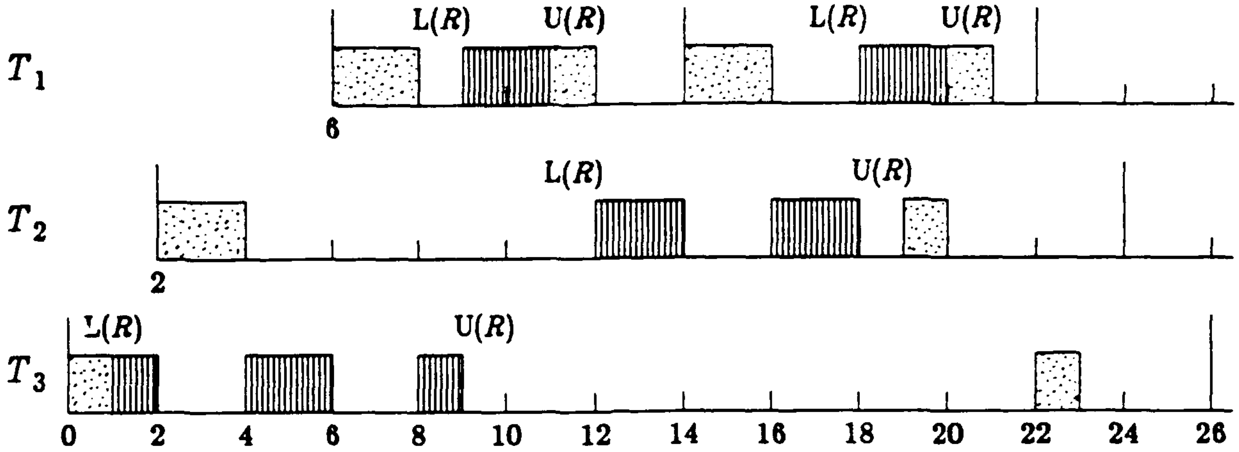
How can we be sure that all tasks are completed in time ?

How about if we

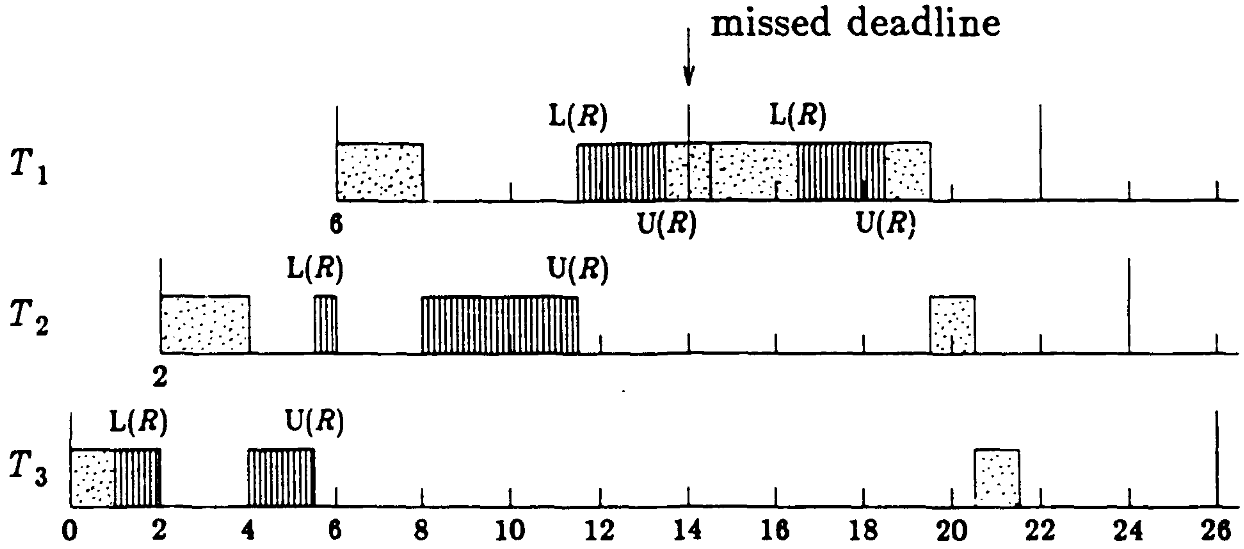
- check whether all deadlines can be met if all tasks have their maximum processing times
- check how much spare time we would have if all tasks have their minimum processing times

But what do these tests tell us ?

Scheduling anomaly

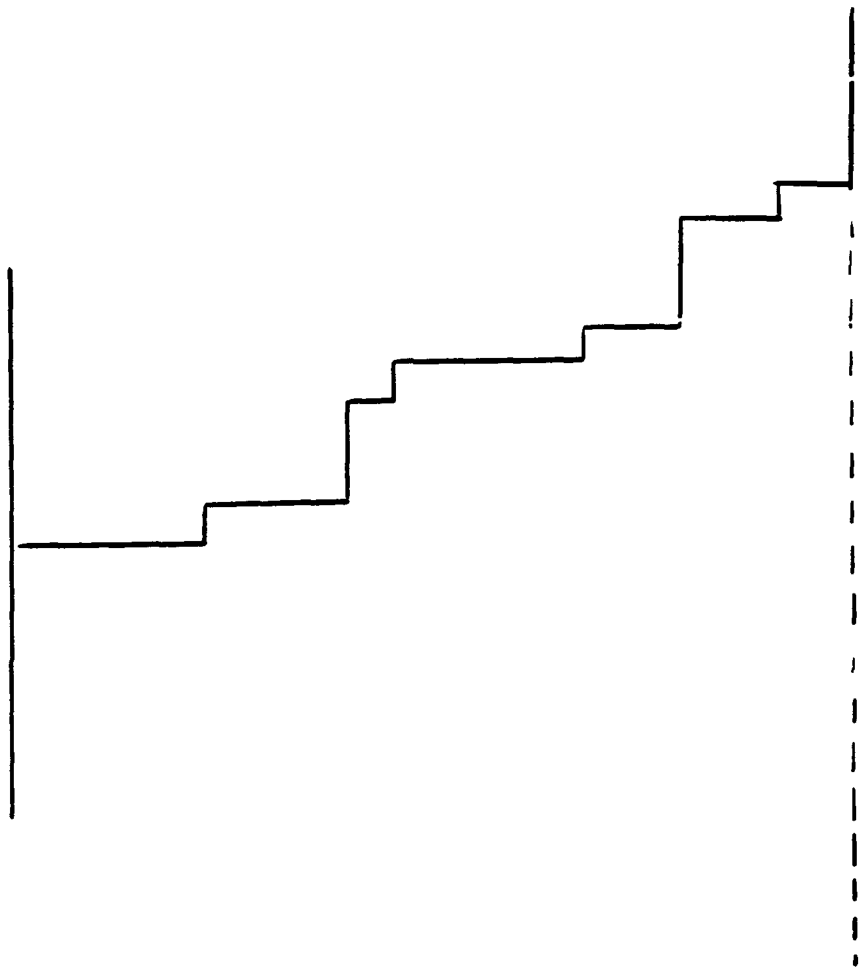


$T_1 = (\ 8,\ 5\ ),\ T_2 = (\ 22,\ 7\ ),\ T_3 = (\ 26,\ 6\ )$

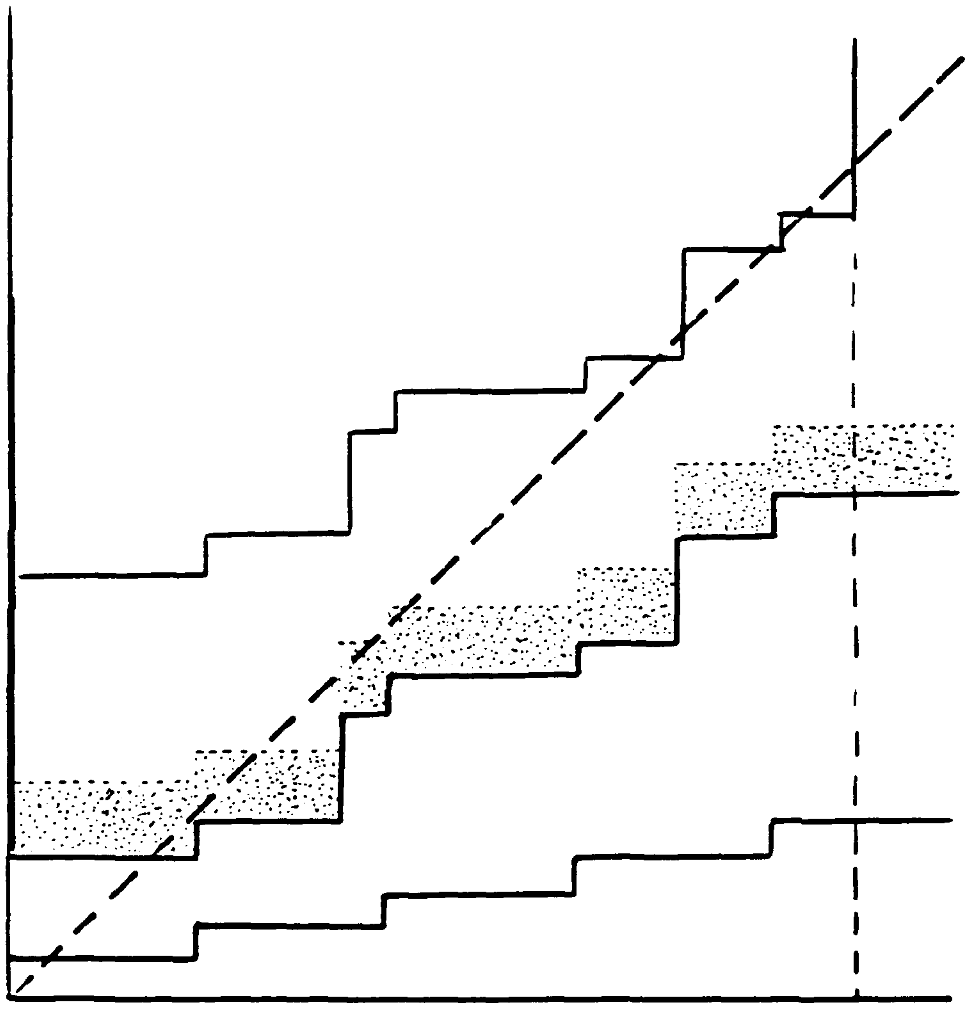


$T_1 = (\ 8,\ 5\ ),\ T_2 = (\ 22,\ 7\ ),\ T_3 = (\ 26,\ 4.5\ )$

We have methods to predict such behavior



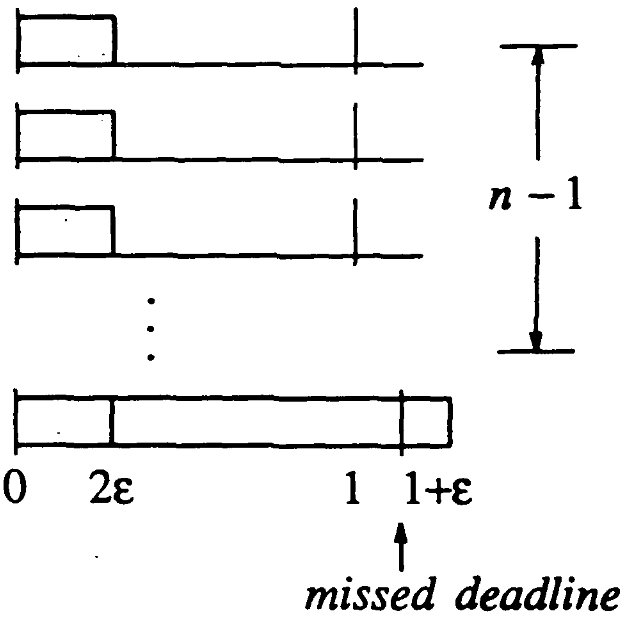
with fixed priorities



Schedulability of periodic tasks
with fixed priorities
on one processor

An Example Illustrating the Unacceptable Performance of the Rate-monotone Algorithm for Multiprocessor Scheduling

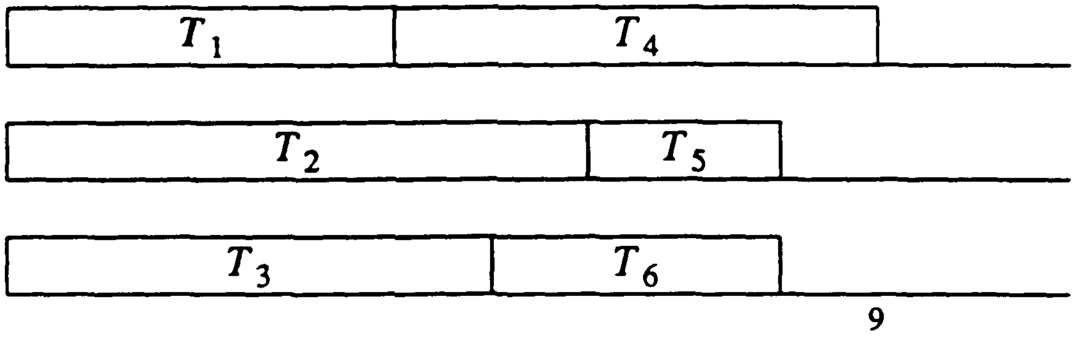
Schedule the $n+1$ jobs $(1, 2\varepsilon), (1, 2\varepsilon), \cdots, (1, 2\varepsilon), (1+\varepsilon, 1)$ on n processors using the rate-monotone algorithm



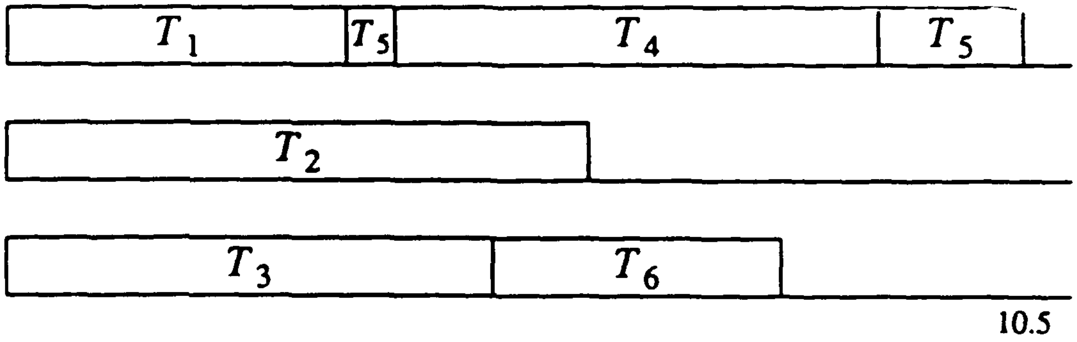
$$U = n\,\frac{2\varepsilon}{1} + \frac{1}{1+\varepsilon} \rightarrow 1$$

Solution: statically bind jobs to processors

6 independent tasks on 3 processors with priority list = $(T_1, T_2, T_3, T_4, T_5, T_6)$

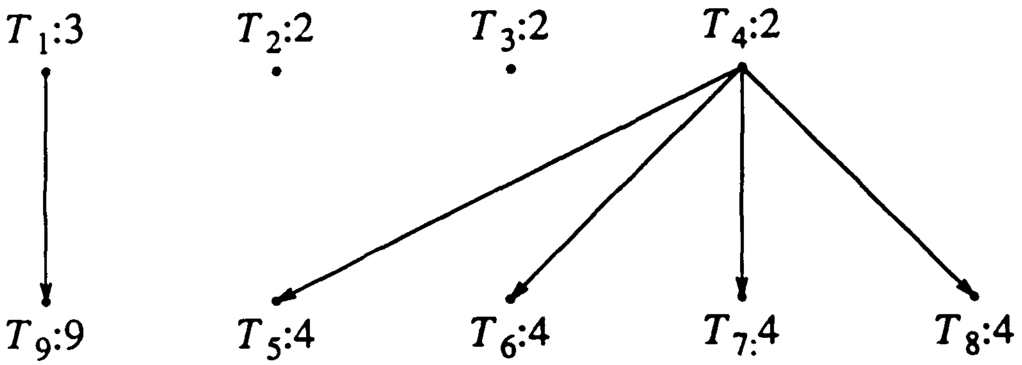


Processing times: 4, 6, 5, 5, 2, 3
Ready Times: 0, 0, 0, 4, 3, 5

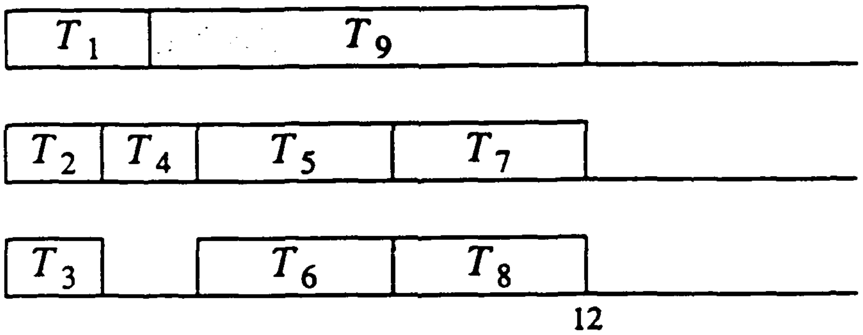


Processing times: 3.5, 6, 5, 5, 2, 3
Ready Times: 0, 0, 0, 4, 3, 5

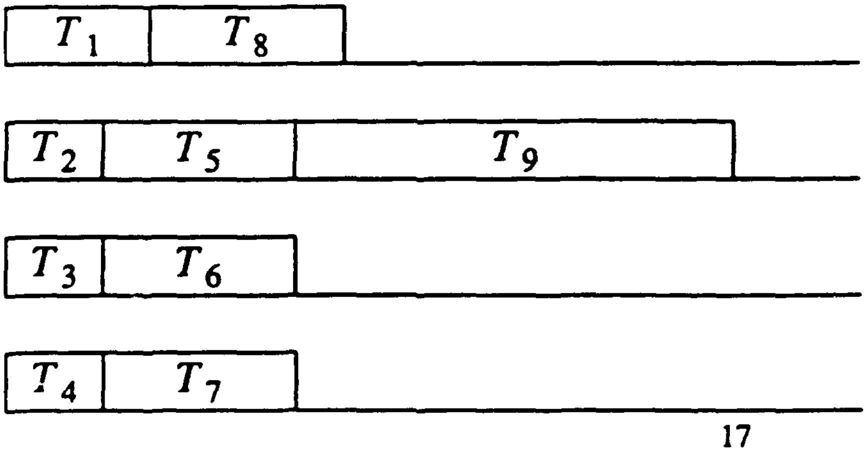
Unexpected Behavior of Priority-Driven Algorithms



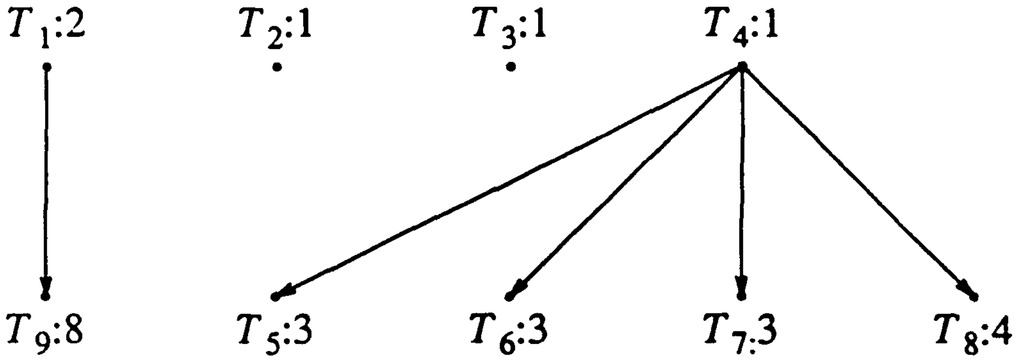
$(T_1, T_2, T_3, T_4, T_5, T_6, T_7, T_8, T_9)$



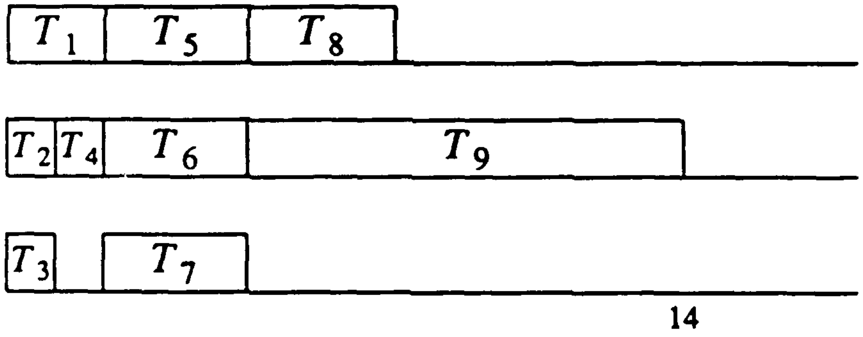
- Suppose that we have four processors instead.



Suppose that tasks have shorter processing times



$(T_1, T_2, T_3, T_4, T_5, T_6, T_7, T_8, T_9)$

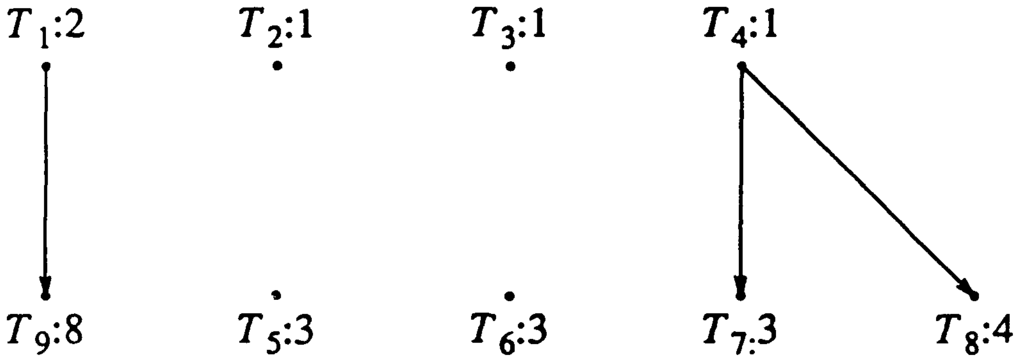


Scheduling to Meet Timing Constraints

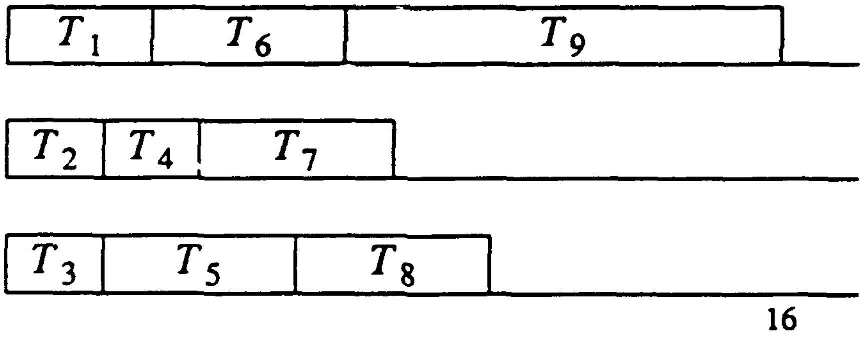
- Remaining problems in the framework of
  - — the periodic-job model and
  - — the complex-job model

- Problems yet to be solved
  - — Scheduling to meet end-to-end deadlines
    - ○ examples from tightly-coupled and loosely-coupled systems
    - ○ variations of the end-to-end scheduling problem
    - ○ related problems, existing solutions, and future work
  - — Dynamic scheduling (and monitor-based scheduling)
    - ○ costs and benefits of dynamic strategies
    - ○ examples of unstable and oscillatory behavior
    - ○ needed solutions, theories and supporting data
  - — Scheduling to enhance dependability
    - ○ scheduling replicated tasks to mask errors
    - ○ scheduling imprecise tasks to increase availability
  - — Scheduling to meet deadlines with high probability
    - ○ model validation and calibration
    - ○ performance profiling techniques, tools and experiment designs

Suppose that tasks are less dependent



$(T_1, T_2, T_3, T_4, T_5, T_6, T_7, T_8, T_9)$



$\omega =$ response time of a set of tasks scheduled according a priority-driven algorithm

$\omega_0 =$ optimal response time

- If tasks are nonpreemptive, independent

$\le \frac{4}{3} - \frac{1}{3m}$ $m =$ no. of processors

- If tasks are dependent, non-preemptive

$\frac{\omega}{\omega_0} \le 2 - \frac{1}{m}$ if processors are identical

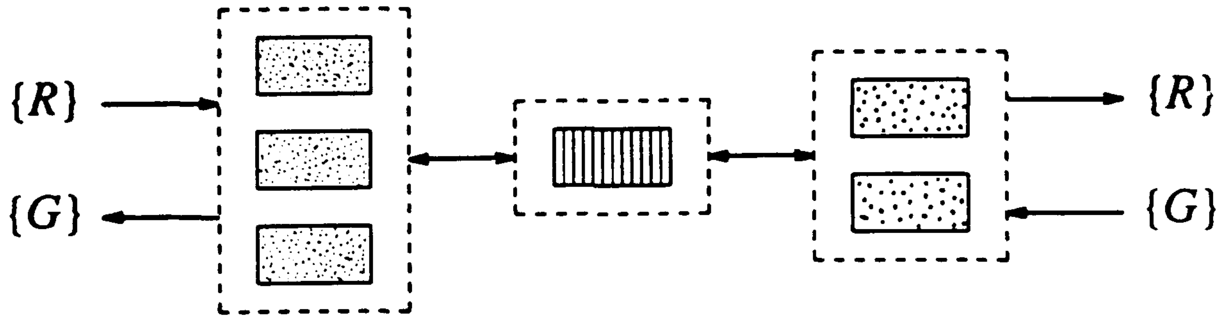
$\le b+1-\frac{Const}{m}$ if the speed ratio is b

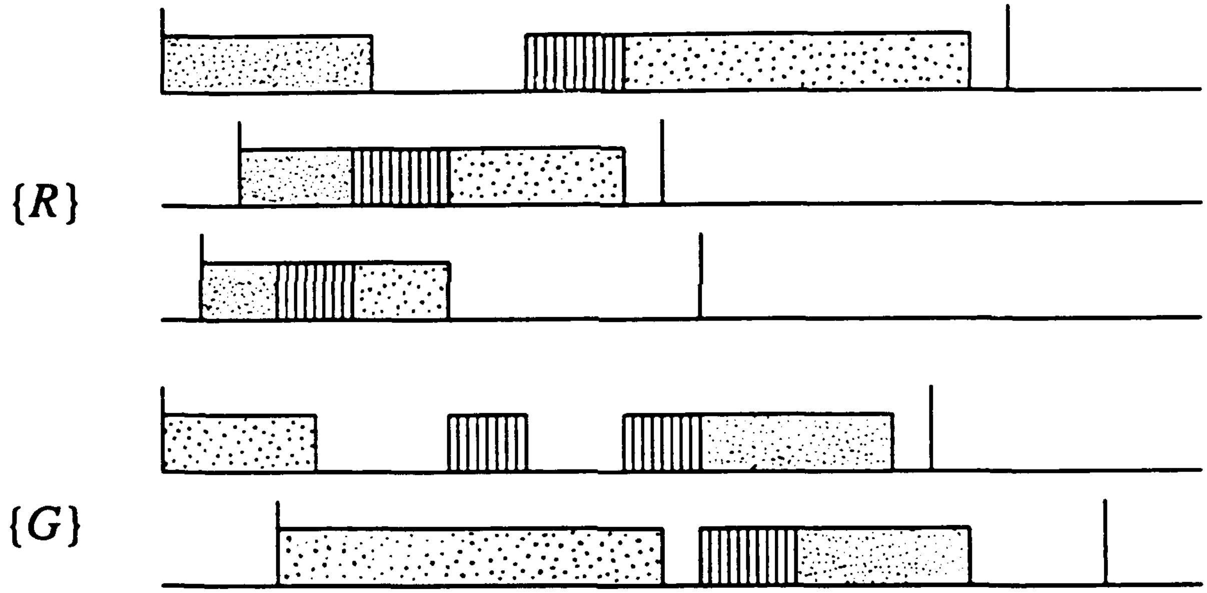
$\le k+1-\frac{const}{m}$ if there are k different kinds of processors all of the same speed

$\le r+1-\frac{const}{m}$ if there are r different kinds of resources

We need similiar bounds for specific task sets and priority-driven algorithm, that is, algorithms for finding the bounds

End-to-End Scheduling





Given:

— a system of physically or functionally distinct types of processors

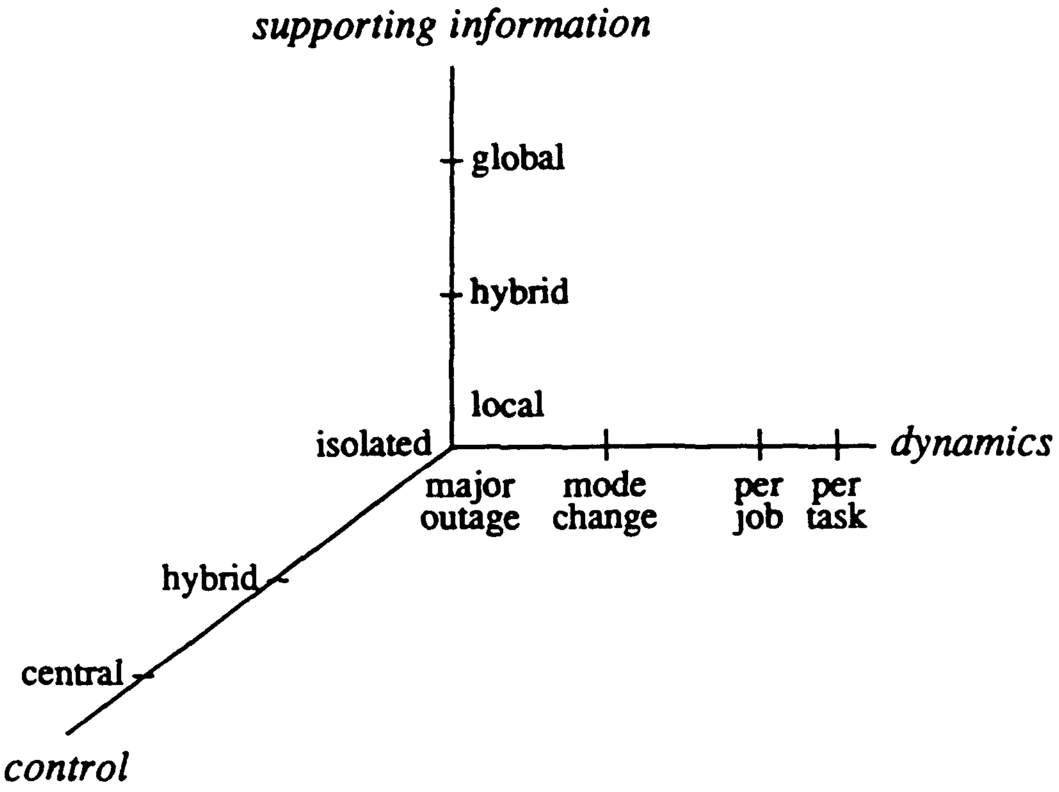
— jobs containing tasks to be executed in turn on different types of processors and having end-to-end deadlines

Find: A schedule meeting end-to-end deadlines whenever such schedules exist.

Past experiences to be avoided

- Instability of CSMA + binary exponential backoff
- Looping problem in old ARPA network (exchange-vector) routing algorithm
- Global failure of the current routing algorithm due to local faults
- Oscillatory behavior and instability of adaptive algorithms

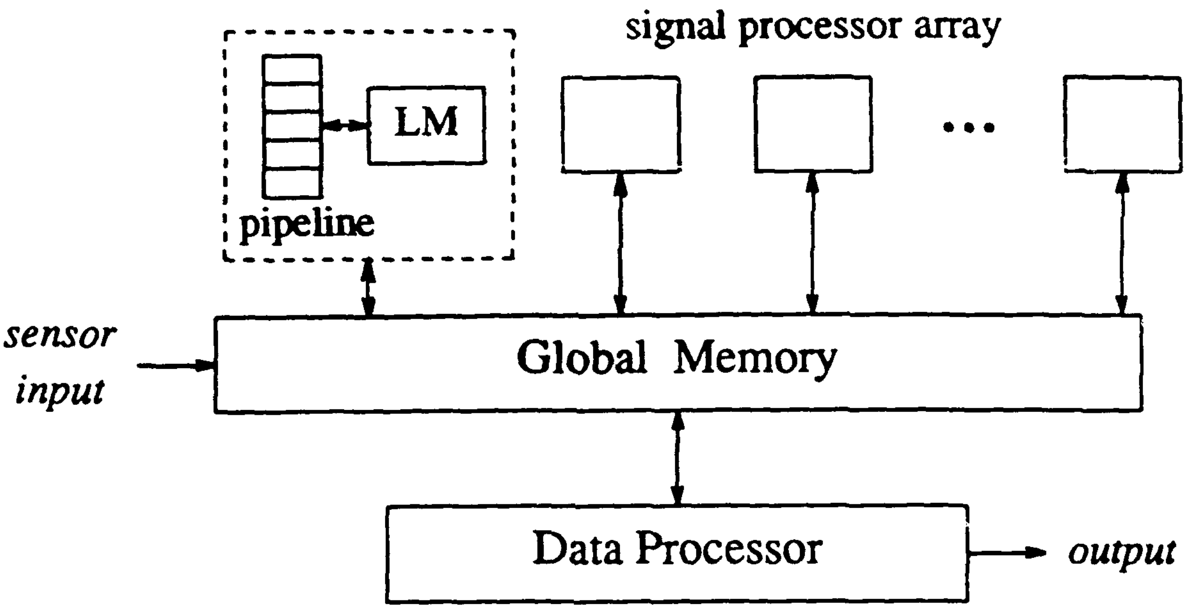
Problems in End-to-end Scheduling

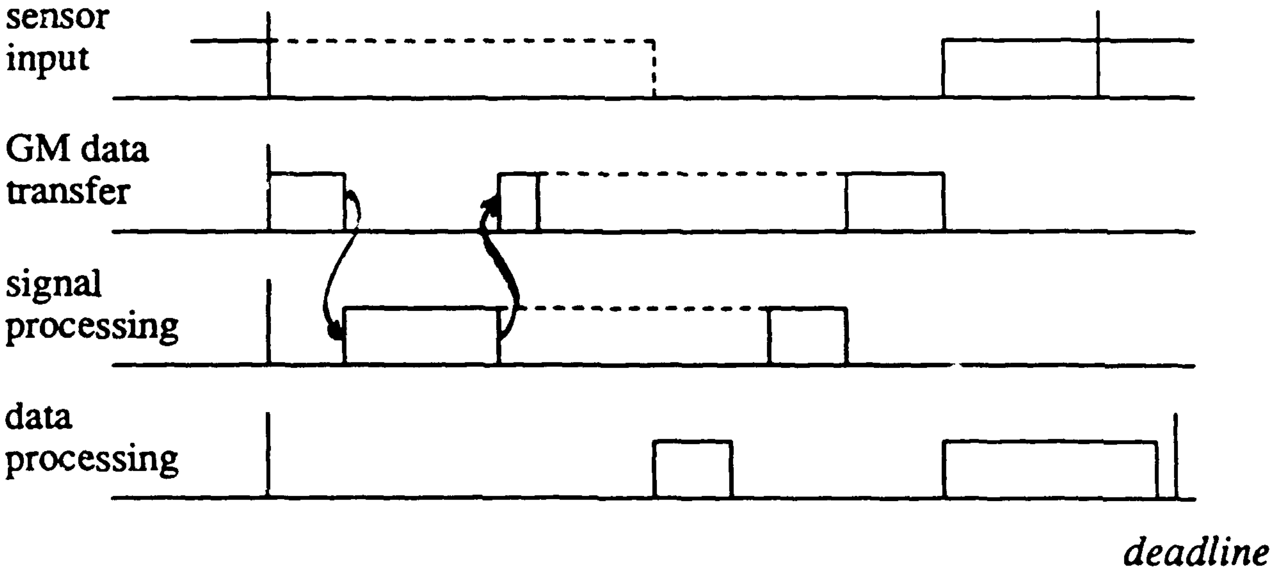


- There are solutions emerging for the cases where
  — global information is available and current;
  — global information is available but may be old, and performance optimization is not important.

- Solutions are needed in all other cases.

End-to-End Scheduling in a Tightly-Coupled System



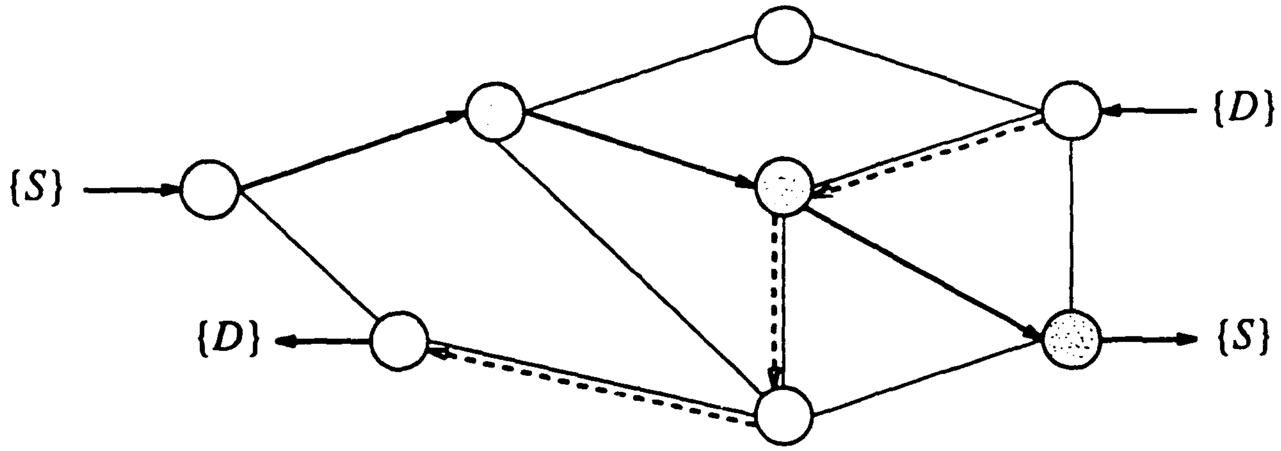


Typical assumptions:

— Global information on load condition and processor status is complete and current.

— Rescheduling is necessary only when mode changes.

End-to-End Scheduling in Loosely-Coupled Environments



- Examples:
  - — Scheduling jobs in remote controllers, command and control systems, process control systems, etc.
  - — Routing and sequencing real-time communications

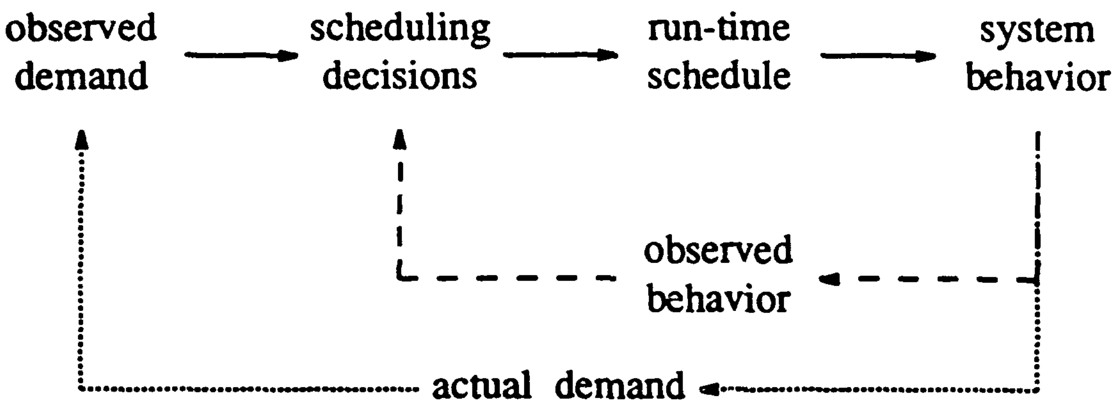
- *Typical assumptions*
  - — Global information is not available, incomplete, or complete but not current.
  - — Scheduling is done
    - o at configuration time and major outage
    - o during mode change
    - o during session (or connection) establishment
    - o on a per job, per task, or per message basis

End-to-End Scheduling without Global Information

- *The only known approach*: first distribute the overall slack time of each job to the individual tasks in it.

- *Components* of the needed solutions include
  - — on-line and nearly-on-line scheduling algorithms (How much can partial and old information help?)
  - — algorithms for scheduling tasks to minimize error or to minimize the number of discarded optional tasks
  - — dynamic and monitor-based algorithms

- What makes an algorithm dynamic?



observed demand → scheduling decisions → run-time schedule → system behavior

observed behavior

actual demand

- Dynamic algorithms are needed
  - — when the demands on the system (and hence the task paramters) are not known completedly or a priori,
  - — when the system must respond to frequent changes in demands or configuration quickly.

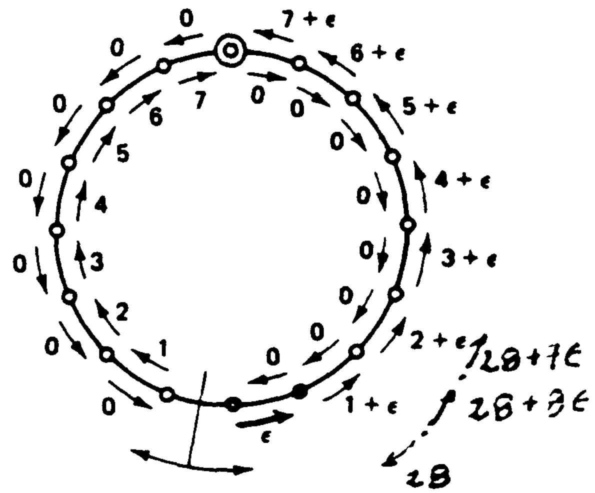
- Examples of dynamic algorithms include
  - — priority-driven algorithms - queue driven
  - — local-balancing algorithms
  - — adaptive algorithms

Critical issues: cost vs benifit, stability and convergence

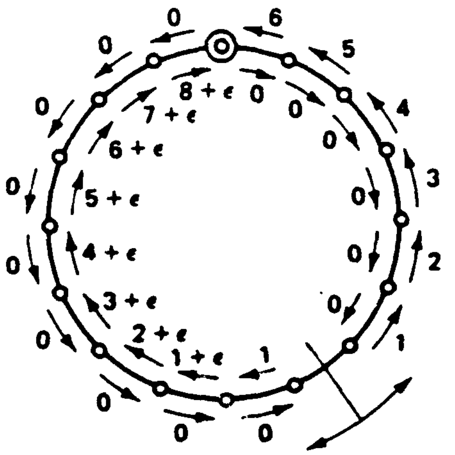
A Stability Issue: Oscillatory Behavior

(An example from *Data Networks* by Bertsekas and Gallager)

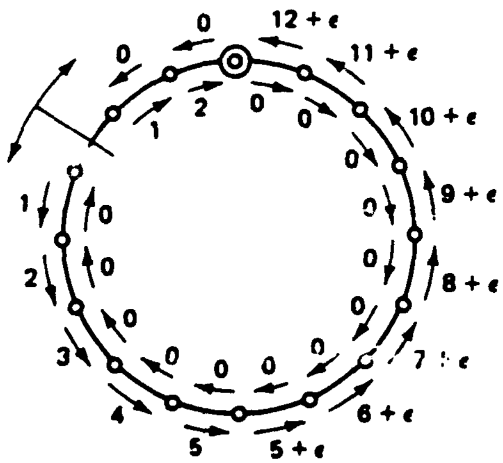
1



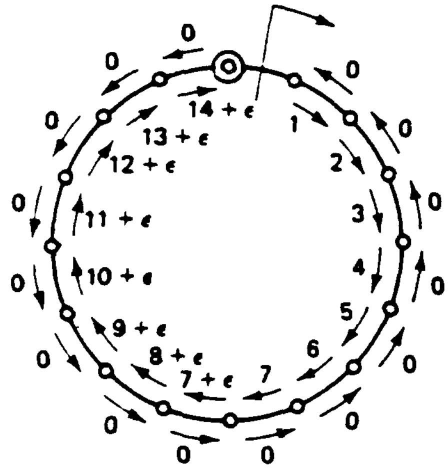
2



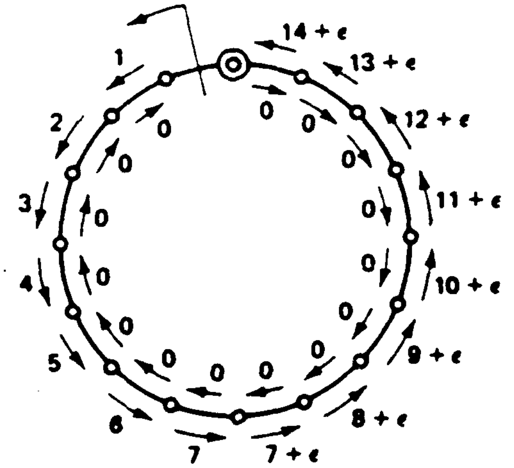
3



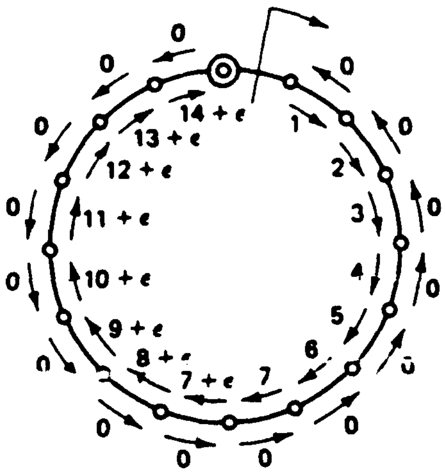
4



5

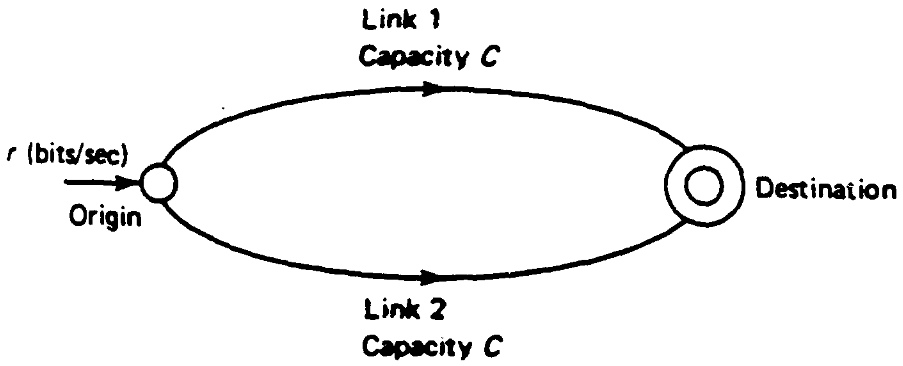


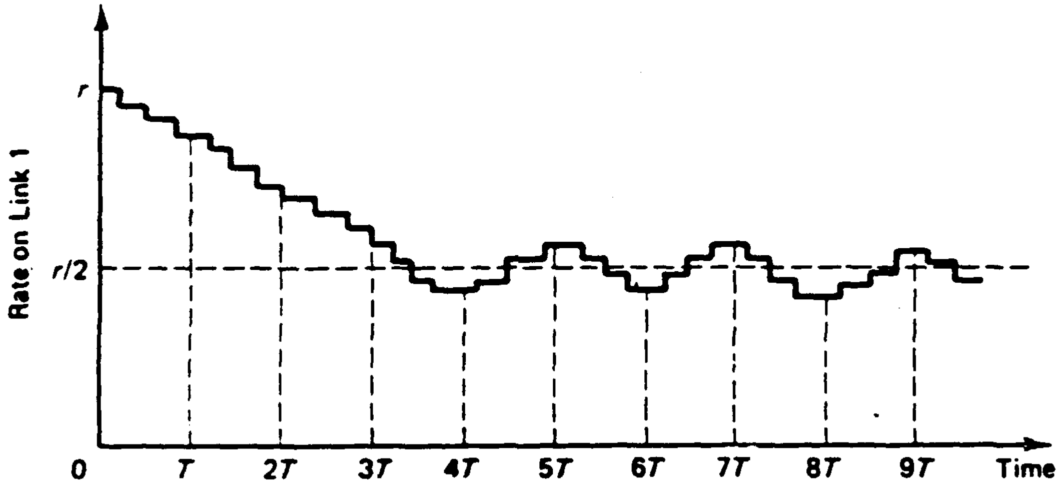
6

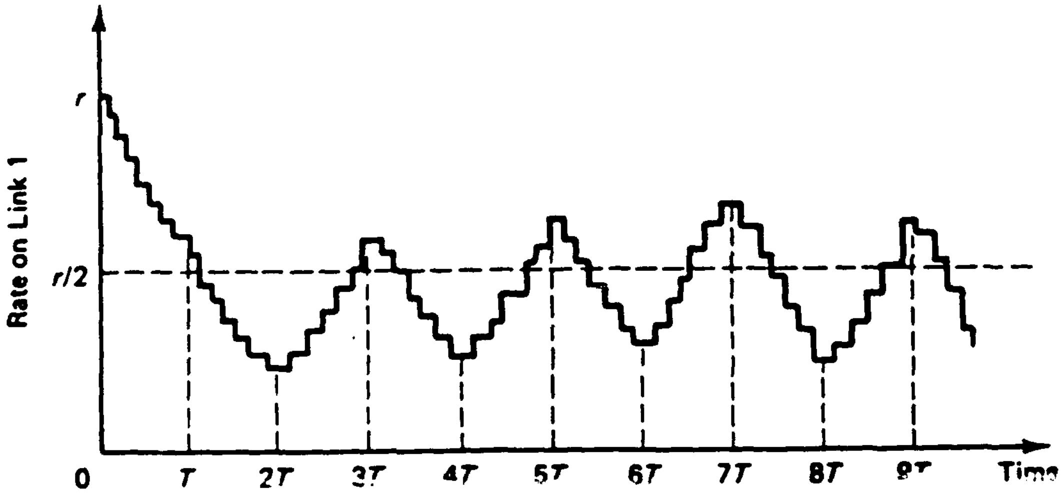


A Stability Issue: Convergence of Adaptive Algorithms

(An example from *Data Networks* by Bertsekas and Gallager)

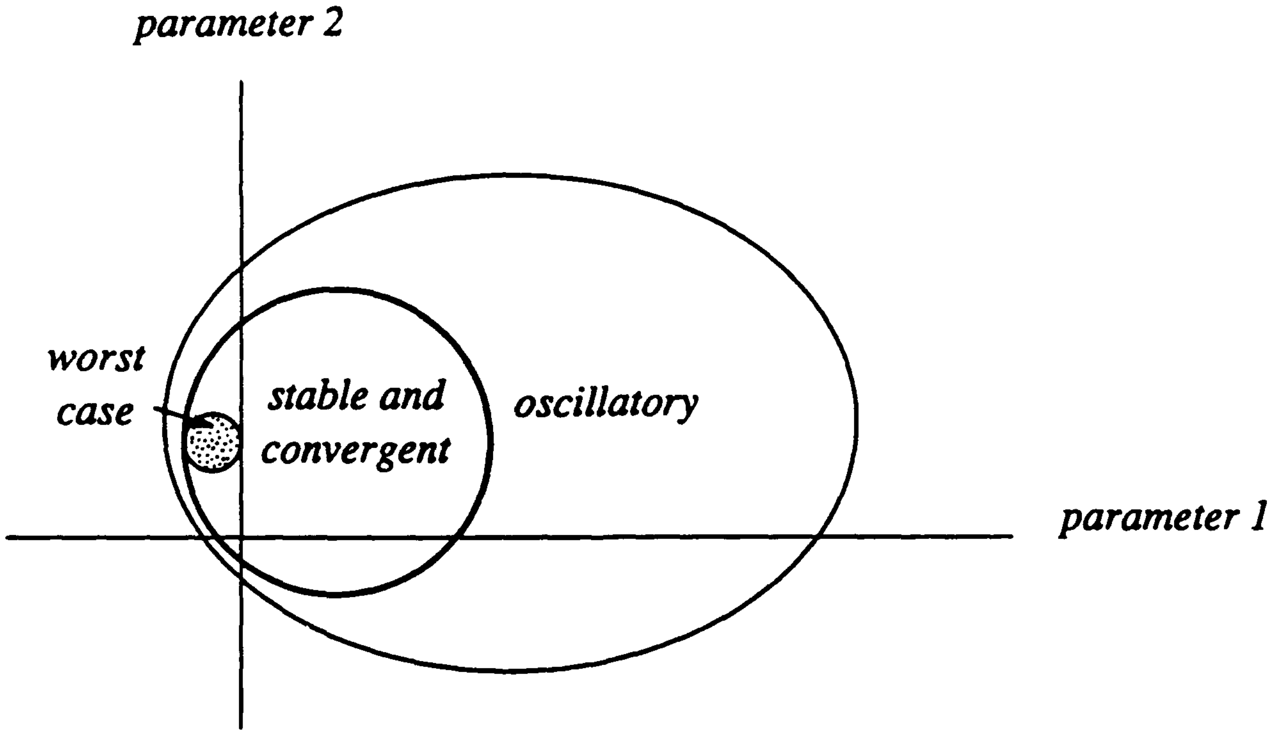






Critical information about a dynamic algorithm ne2ded to support its safe usage

- — cost vs benifit
- — regions of stability and oscillatory behavior
- — ideal operating region and parameters tunable to keep the operating point in the region
- — worst-case operating region



Validation and Verification of Real-Time Systems Developments

March, 1993

C. Douglass Locke

IBM Federal Systems Company
Bethesda, MD, USA
locke@vnet.ibm.com
(301) 493-1496

Outline

Introduction

Prerequisites to Verification and Validation

Verification and Validation

Conclusions

Introduction

Definitions

- Validation - Determination that the solution can be made to work as specified.
- Verification - Determine that the solution works as specified.

Prerequisites to Verification and Validation of Real-Time Systems

- Timing & Performance Requirements
- Analyzable Architecture
- Resource Usage Estimates (processing load, network load, I/O rates)
- Measurement Methodology
- Analysis Tools

Verification: not done only following implementation

- Continuous process.
- Not just something to do at the end of the implementation.
- Not to be confused with an acceptance test.

Timing & Performance Requirements

Timing and performance requirements are generally derived requirements

- Actual system requirements leading to timing and performance requirements are, e.g., accuracy, availability, human responsiveness.
- Many system requirement specifications omit timing and performance requirements.
- When expressed, they are almost always end-to-end requirements.

Analyzable Software and Systems Architecture

Architecture is the set of high level design decisions defining:

- Processors
- Communications
- Concurrency
- High-level data definition (generally as sets).

Time constraints must drive the software and systems architecture.

- Operating Systems
- Communications protocols
- Languages
- Databases
- GUI's

Resource Usage Estimates

All resources must be considered, including:

- Processing load
- Network load
- I/O rates

Estimates must be made for each schedulable entity:

- Tasks
- Processes
- Threads

Timing requirements must be decomposed, resulting in:

- Periodicity
- Aperiodic arrival rates
- Aperiodic interarrival times

Measurement Methodology

Some means for measuring resource usage required:

- Operating System trace functions
- Communications monitors
- I/O recording functions
- ICE hardware
- Logic Analyzers
- Application-level measurements

Analysis Tools

Dependent on scheduling/architectural models used, e.g.:

- RMA tools
- Composite temporal merit computation

Verification and Validation

Continuous development life-cycle activity:

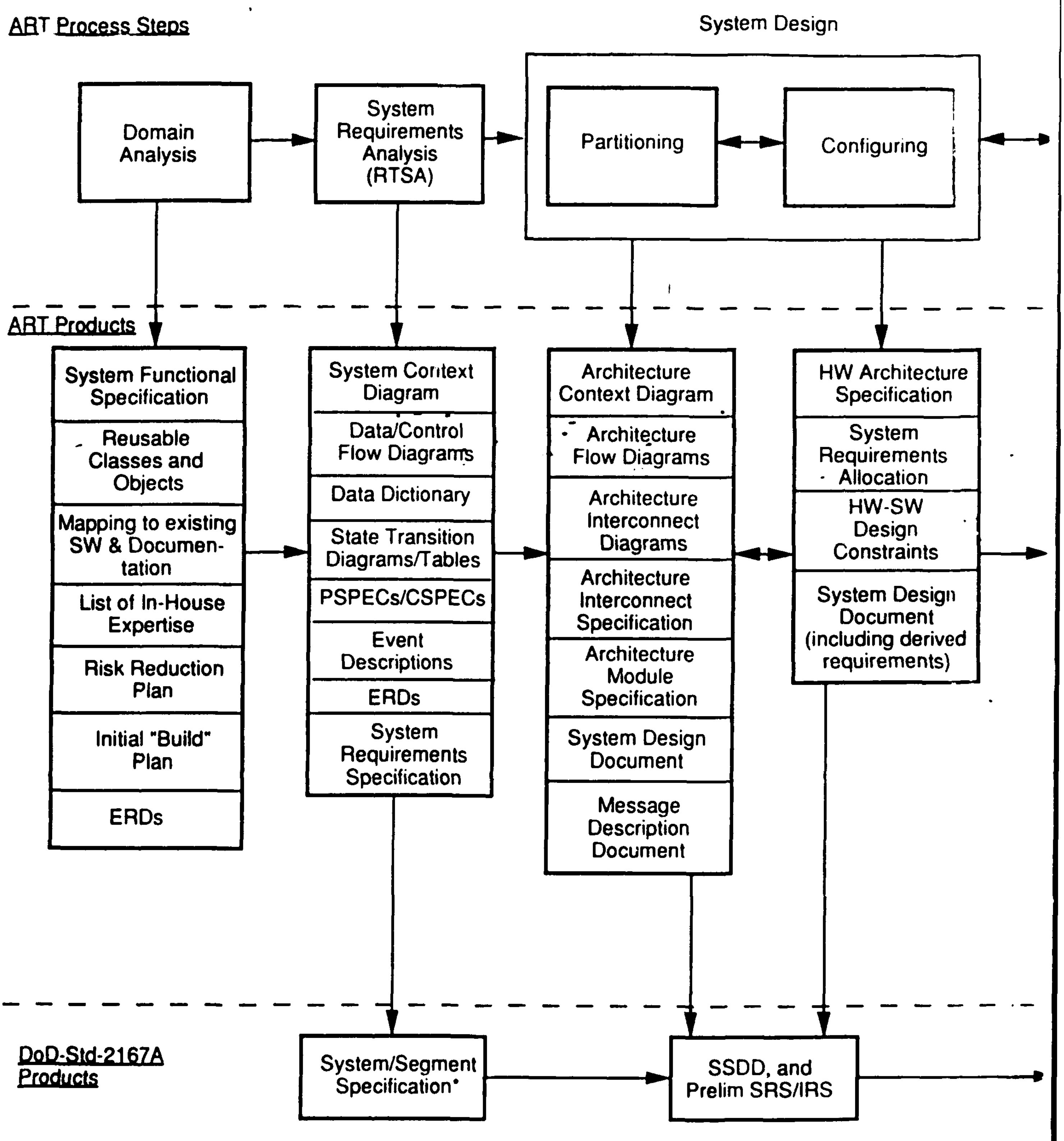
- Architecture Verification - Determination that the architecture can meet specified performance and timing constraints.
- Design Verification - Determination that the design implements the architecture, and will thus meet the specified performance and timing constraints.
- Code Inspection - Formal or informal determination that the coded software implements the design, and will thus meet the specified performance and timing constraints.
- Model Validation - Measurement of implementation components that directly address archecture, and will thus meet the specified performance and timing constraints.
- Performance Verification - Demonstration that the finished product meets all externally visible specifications, including functional, accuracy, availability, and timing.

Conclusions

Real-time validation and verification fundamentally dependent on:

- Early determination of architecture, derived timing requirements, and load
- Continuous tracking of development against estimates
- Early resource utilization contingency management

In short, resource management for real-time must be managed in the same way
as cost.


ART Process Steps
System Design
Domain Analysis
System Requirements Analysis (RTSA)
Partitioning
Configuring
ART Products
System Functional Specification
Reusable Classes and Objects
Mapping to existing SW & Documentation
List of In-House Expertise
Risk Reduction Plan
Initial "Build" Plan
ERDs
System Context Diagram
Data/Control Flow Diagrams
Data Dictionary
State Transition Diagrams/Tables
PSPECs/CSPECs
Event Descriptions
ERDs
System Requirements Specification
Architecture Context Diagram
Architecture Flow Diagrams
Architecture Interconnect Diagrams
Architecture Interconnect Specification
Architecture Module Specification
System Design Document
Message Description Document
HW Architecture Specification
System Requirements Allocation
HW-SW Design Constraints
System Design Document (including derived requirements)
DoD-Std-2167A Products
System/Segment Specification*
SSDD, and Prelim SRS/IRS

*Mil-Std-490A
Type A Spec

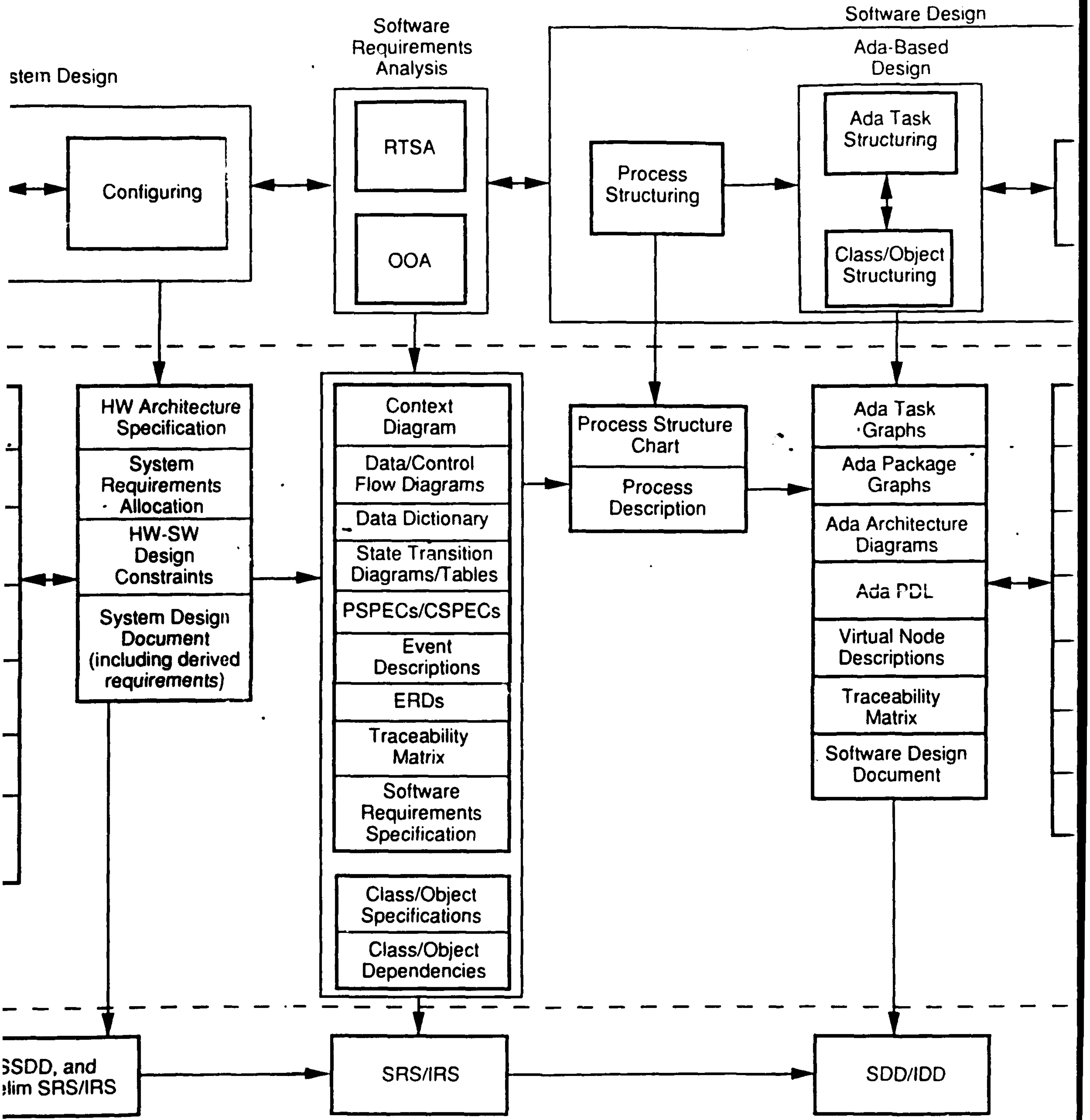
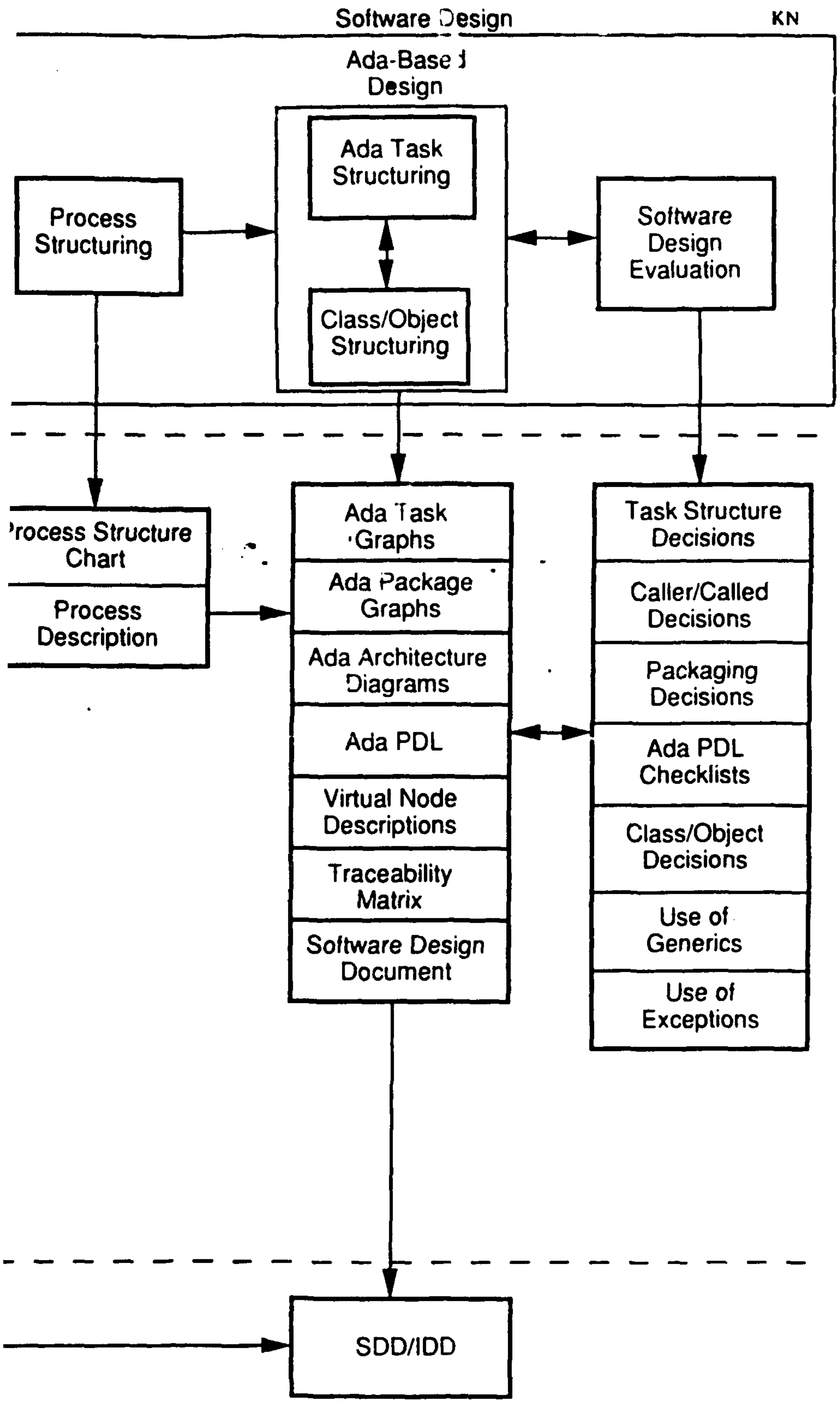


Figure 1-1. ART Process Steps and Products, and 490A/2167A Products



ART Process Steps and Products, 67A Products

HUGHES

v 1.2, Dec 92

3

IDA WORKSHOP
ON

LARGE, DISTRIBUTED, PARALLEL ARCHITECTURE, REAL-TIME SYSTEMS

MARCH 15-18, 1993

KJELL NIELSEN
Aerospace & Defense Sector
Software Engineering Division
(714)732-3849

OVERVIEW

HUGHES

- → THE SYSTEM DEVELOPMENT PROCESS
- REAL-TIME DESIGN AND SCHEDULING THEORY
- VALIDATING LARGE, DISTRIBUTED, REAL-TIME SYSTEMS
- IMPROVING THE DEVELOPMENT PROCESS
- SUMMARY

IDA, 3/93, K. Nielsen

Page 2

THE SYSTEM DEVELOPMENT PROCESS

- → DEVELOPMENT APPROACHES
- DEVELOPMENT STEPS
- INCORPORATING METHODOLOGIES
- OBJECT TECHNOLOGY

DEVELOPMENT APPROACHES (1)

- WATERFALL
  - RIGID
  - WELL-DOCUMENTED
  - LITTLE INTERACTION BETWEEN STEPS
- PROTOTYPING
  - SUPPORTS DEM-VAL/OPS CONCEPTS
  - SOMETIMES ASSOCIATED WITH “THROW-AWAY” DESIGN AND CODE
- ITERATIVE (SPIRAL)
  - DESIGN A LITTLE, CODE A LITTLE, ...
- INCREMENTAL
  - “BUILD” ORIENTED
  - MAY INCORPORATE THE OTHER APPROACHES FOR EACH BUILD

DEVELOPMENT APPROACHES (2)

HUGHES

- A MODERN SYSTEM DEVELOPMENT PROCESS MUST SUPPORT ALL OF THE VARIOUS APPROACHES
    - INCORPORATE METHODS THAT FIT A GIVEN APPROACH
    - DEVELOPERS PICK FROM "TOOL KIT"
    - MAINTAIN REASONABLE BALANCE BETWEEN REQUIRED DOCUMENTATION AND DEVELOPMENT PRODUCTS
- A SYSTEM DEVELOPMENT APPROACH FOR SOFTWARE AND HARDWARE

THE SYSTEM DEVELOPMENT PROCESS

HUGHES

- DEVELOPMENT APPROACHES
- → DEVELOPMENT STEPS
- INCORPORATING METHODOLOGIES
- OBJECT TECHNOLOGY

DEVELOPMENT STEPS

- ART IS ILLUSTRATED IN FIGURE 1-1
  - SUPPORTS SYSTEM DEVELOPMENT
  - INCLUDES DOMAIN ANALYSIS

- INCLUDES THE FOLLOWING MAJOR STEPS
  - DOMAIN ANALYSIS
  - SYSTEM REQUIREMENTS ANALYSIS
  - SYSTEM DESIGN
  - SOFTWARE REQUIREMENTS ANALYSIS
  - SOFTWARE DESIGN

DOMAIN ANALYSIS

HUGHES

- ANALYSIS OF PROBLEM DOMAIN
- IDENTIFY REAL-WORLD CLASSES AND OBJECTS
- SUPPORTS REUSABILITY-IN-THE-LARGE
  - ENTIRE SUBSYSTEMS
  - DESIGN, CODE, AND DOCUMENTATION
  - EXPERTISE
- CREATE RISK REDUCTION PLAN
- IDENTIFY “BUILD” STRUCTURE
- ESPECIALLY IMPORTANT FOR CREATION OF PRODUCT LINE

SYSTEM REQUIREMENTS ANALYSIS

- SUBSTEPS [SHU92]:
  - ESTABLISH SYSTEM OBJECTIVES
  - IDENTIFY FUNCTIONS AND FUNCTION INTERACTIONS
  - SPECIFY THE PERFORMANCE FOR EACH FUNCTION
  - DEFINE AN OVERALL OPERATIONAL CONCEPT
  - ITERATE AS REQUIRED

- PRODUCT
  - SYSTEM REQUIREMENTS SPECIFICATION

- PRECURSOR TO SYSTEM DESIGN

SYSTEM DESIGN (1)

HUGHES

- ALLOCATION OF SYSTEM REQUIREMENTS TO HARDWARE & SOFTWARE
- PARTITION SYSTEM REQUIREMENTS INTO SUBSYSTEMS
- CONFIGURE SUBSYSTEM REQUIREMENTS TO HARDWARE COMPONENT AND SOFTWARE COMPONENT
  - MAY HAVE TO ITERATE TO GET BEST PARTITION AND CONFIGURATION
- HARDWARE COMPONENTS BECOME PHYSICAL NODES (PNs) IN A DISTRIBUTED SYSTEM
- SOFTWARE COMPONENTS ARE LATER DESIGNED (POSSIBLY COMBINED WITH OTHER COMPONENTS OR REPLICATED) AS VIRTUAL NODES (VNs)
- VNs ARE EVENTUALLY MAPPED TO PNs

SYSTEM DESIGN (2)

HUGHES

- WE STRIVE FOR CONCURRENT ENGINEERING
  - SYSTEMS APPROACH
  - HARDWARE AND SOFTWARE ENGINEERS COMMUNICATING
- HARDWARE-FIRST APPROACH
  - HARDWARE ARCHITECTURE DETERMINED W/O CONCERN FOR SOFTWARE ISSUES
- SOFTWARE-FIRST APPROACH
  - SOFTWARE REQUIREMENTS ANALYSIS WITH SYSTEM VIEW
  - INDEPENDENT OF HARDWARE ARCHITECTURE
  - "SYSTEM DESIGN" PERFORMED LATER
  - MUST CONSIDER DISTRIBUTED ISSUES EARLY
  - USEFUL FOR DEVELOPING PRODUCT LINE

SYSTEM DESIGN (3)

HUGHES

- PROBLEM AREAS
  - WHEN DO WE START TO CONSIDER DISTRIBUTED ISSUES?
  - DO WE WAIT UNTIL THE SOFTWARE REQUIREMENTS ANALYSIS TO CONSIDER AN IPC MECHANISM?
  - DO WE KNOW ENOUGH ABOUT THE REQUIREMENTS TO DECIDE ON
    - SYNCH VS ASYNCH MESSAGE PASSING?
    - BROADCAST AND MULTICAST?

SOFTWARE REQUIREMENTS ANALYSIS (1)

HUGHES

- START OF SOFTWARE DEVELOPMENT FOR EACH SUBSYSTEM
  - SOFTWARE ASSOCIATED WITH A "BLACK BOX"
  - CSCI FOR 2167A
  - SOFTWARE COMPONENT FOR SOFTWARE-FIRST
- USING REAL-TIME STRUCTURED ANALYSIS (RTSA)
  - HATLEY-PIRBHAI [HAT87]
  - WARD-MELLOR [WAR85, MEL86]
- OBJECT-ORIENTED ANALYSIS (OOA)
  - REAL-WORLD CLASSES AND OBJECTS (STATIC MODEL)
  - CLASS/OBJECT RELATIONS (DYNAMIC MODEL)
  - WHICH METHOD DO WE PICK?
  - WHERE IS DISTINCTION BETWEEN ANALYSIS AND DESIGN?

SOFTWARE REQUIREMENTS ANALYSIS (2)

HUGHES

- PROBLEM AREAS
  - PROPER USE OF RTSA AND OOA
  - WHICH OOA METHOD TO USE
    - NO STANDARD OOA NOTATION
    - STATIC AND DYNAMIC MODELS
    - EACH OOA METHOD WITH ITS OWN TOOL
  - STOPPING CRITERIA FOR “ANALYSIS” (BEFORE “DESIGN”)

SOFTWARE DESIGN (1)

HUGHES

- TRANSITIONING FROM ANALYSIS TO DESIGN
- PROCESS STRUCTURING (CONCURRENCY)
- LANGUAGE-DEPENDENT DESIGN
  - SOFTWARE ARCHITECTURE
  - CLASS/OBJECT STRUCTURING
  - ITERATION IS EXPECTED
- CREATION OF VIRTUAL NODES (AND MAPPING TO PNs)
- DESIGN EVALUATION
  - INFORMAL REVIEWS
  - FORMAL REVIEWS
  - DESIGN GUIDELINES
  - STARTS WITH INITIAL PROTOTYPE
  - CONTINUES THROUGH FINAL BUILD

SOFTWARE DESIGN (2)

HUGHES

- PROBLEM AREAS
  - TRANSITIONING FROM ANALYSIS TO DESIGN
  - DETERMINATION OF CONCURRENCY
  - DISTRIBUTED ISSUES
    - IPC MECHANISM
    - DESIGN RESTRICTIONS
    - LANGUAGE DEPENDENCE
  - USE OF OOD
    - WHICH METHOD
    - NOTATION
    - CONCURRENCY
    - DISTRIBUTED ISSUES

THE SYSTEM DEVELOPMENT PROCESS

- DEVELOPMENT APPROACHES
- DEVELOPMENT STEPS
- → INCORPORATING METHODOLOGIES
- OBJECT TECHNOLOGY

INCORPORATING METHODOLOGIES (1)

HUGHES

- ART INCORPORATES SEVERAL METHODOLOGIES
  - RTSA
    - SYSTEM REQUIREMENTS ANALYSIS
    - SYSTEM DESIGN
    - SOFTWARE REQUIREMENTS ANALYSIS
  - OOA/OOD
    - DOMAIN ANALYSIS
    - SOFTWARE REQUIREMENTS ANALYSIS
    - SOFTWARE DESIGN
  - DARTS
    - TRANSITIONING FROM ANALYSIS TO DESIGN
  - DATA MODELING (ERDs)
    - DOMAIN ANALYSIS
    - SYSTEM REQUIREMENTS ANALYSIS
    - SOFTWARE REQUIREMENTS ANALYSIS

INCORPORATING METHODOLOGIES (2)

HUGHES

- PROBLEM AREAS
  - MATCHING METHODS AND PROBLEM DOMAINS
  - INTRODUCING NEW METHODS WITH MINIMAL IMPACT
  - CAPITALIZING ON EXISTING AND PROVEN METHODS
  - MAINTAINING CONSISTENT NOTATION ACROSS METHODS
  - FINDING METHODS THAT ADDRESS CONCURRENCY AND DISTRIBUTED ISSUES
  - FINDING INTEGRATED CASE TOOLS

THE SYSTEM DEVELOPMENT PROCESS

HUGHES

- DEVELOPMENT APPROACHES
- DEVELOPMENT STEPS
- INCORPORATING METHODOLOGIES
- → OBJECT TECHNOLOGY

IDA, 3/93, K. Nielsen

Page 20

OBJECT TECHNOLOGY (OT)

HUGHES

- LOTS OF CONTROVERSY FOR LARGE, REAL-TIME SYSTEMS
  - STRICTLY BOTTOM-UP
  - MIX OF BOTTOM-UP AND TOP-DOWN
  - "ORTHOGONALITY" OF OT, RTSA, AND ART (FUNCTIONAL METHODS)

  - TREATMENT OF CONCURRENCY
  - DISTRIBUTED ISSUES
  - EXPENSIVE TO INTRODUCE AND IMPLEMENT
  - WHICH OOX METHOD DO WE CHOOSE?

- OT IS IMPORTANT PART OF SYSTEM DEVELOPMENT APPROACH
  - OT, RTSA, AND ART COMPLEMENT EACH OTHER

OVERVIEW

- THE SYSTEM DEVELOPMENT PROCESS
- → REAL-TIME DESIGN AND SCHEDULING THEORY
- VALIDATING LARGE, DISTRIBUTED, REAL-TIME SYSTEMS
- IMPROVING THE DEVELOPMENT PROCESS
- SUMMARY

REAL-TIME DESIGN AND SCHEDULING THEORY (1)

HUGHES

- INHERENT IN LARGE DISTRIBUTED SYSTEMS
  - CONCURRENCY
    - REAL (MULTIPLE CPUs)
    - APPARENT (COMPETITION FOR A CPU)
    - MUTUAL EXCLUSION
  - REQUIRES EFFICIENT RUN-TIME SCHEDULING
    - PREEMPTIVE
    - NON-PREEMPTIVE
    - "FAIR"
  - DESIGNERS MUST CONSIDER SCHEDULING MECHANISM
    - TIME CRITICALITY
    - PERFORMANCE REQUIREMENTS
    - COMMUNICATION BETWEEN PROCESSES
  - EFFICIENT IPC MECHANISM

HUGHES

REAL-TIME DESIGN AND SCHEDULING THEORY (2)

- PROBLEM AREAS
  - DESIGN METHODOLOGISTS AND SCHEDULING THEORISTS HAVE NOT COMMUNICATED WELL
  - ADA'S SCHEDULING MODEL HAS LIMITATIONS
  - OTHER PROGRAMMING LANGUAGES HAVE NO SCHEDULING MECHANISM (MUST USE OS SERVICES)
  - DISTRIBUTED AND CONCURRENT DESIGNS ARE HIGHLY DEPENDENT ON AVAILABLE SCHEDULING MECHANISMS
  - REAL-TIME SCHEDULING NOT WELL UNDERSTOOD BY DESIGNERS
  - NO STANDARD IPC MECHANISM AVAILABLE

IDA, 3/93, K. Nielsen

Page 24

OVERVIEW

- THE SYSTEM DEVELOPMENT PROCESS
- REAL-TIME DESIGN AND SCHEDULING THEORY
- → VALIDATING LARGE, DISTRIBUTED, REAL-TIME SYSTEMS
- IMPROVING THE DEVELOPMENT PROCESS
- SUMMARY

VALIDATING LARGE, REAL-TIME, DISTRIBUTED SYSTEMS (1)

HUGHES

- VALIDATION INCLUDES THREE PRIMARY ELEMENTS
  - PLANS AND PROCEDURES FOR VALIDATION
  - CONSISTENCY IN IMPLEMENTING PLANS AND PROCEDURES
  - MODELING AND TEST TOOLS TO SUPPORT VALIDATION

- 2167A ACTUALLY HELPS HERE!
  - DIDs SUPPORT LEVELS OF PLANNING AND IMPLEMENTATION

- VALIDATION CONTINUES THROUGHOUT THE DEVELOPMENT
  - ATTEMPT TO FIND DESIGN ERRORS AND BUGS AS EARLY AS POSSIBLE
  - PROTOTYPING
  - INCREMENTAL DEVELOPMENT
  - FINAL SYSTEM TESTING

VALIDATING LARGE, REAL-TIME, DISTRIBUTED SYSTEMS (2)

HUGHES

- PROBLEM AREAS
    - NO STANDARD PROCEDURE FOR VALIDATION
        - METRICS
        - ERROR REPORTING
    - DIFFICULT TO MEASURE PERFORMANCE ACROSS MULTIPLE PROCESSORS
    - SYSTEM TESTING IS SOMETIMES AD HOC
        - CONCENTRATING ON GETTING THE SYSTEM TO RUN
    - AUTOMATED TEST TOOLS ARE LIMITED

OVERVIEW

- THE SYSTEM DEVELOPMENT PROCESS
- REAL-TIME DESIGN AND SCHEDULING THEORY
- VALIDATING LARGE, DISTRIBUTED, REAL-TIME SYSTEMS
- → IMPROVING THE DEVELOPMENT PROCESS
- SUMMARY

IMPROVING THE DEVELOPMENT PROCESS (1)

HUGHES

- ESTABLISH DESIGN GUIDELINES FOR IPC MECHANISM
  - RPC
  - MESSAGE PASSING
  - BROADCAST, MULTICAST, CONNECTION-ORIENTED, ...
  - SYNCHRONOUS, ASYNCHRONOUS, ...
  - SHARED DATA, HETEROGENEOUS, HOMOGENEOUS, ...
  - TCP/IP, OSI
  - C/C++/ADA BINDINGS

- ESTABLISH DESIGN GUIDELINES FOR DISTRIBUTED DATABASES
  - STANDARD LOCKING MECHANISMS

- IMPROVE EXISTING CASE TOOLS

- RECOMMEND OOA/OOD METHOD FOR DISTRIBUTED SYSTEMS

IMPROVING THE DEVELOPMENT PROCESS (2)

- EXPAND EXISTING SCHEDULING MECHANISMS
  - ADA
  - C/C++
- DEVELOP PROTOTYPING TOOLS
  - X/MOTIF, OSI, TCP/IP
  - C/C++/ADA
- EXPAND PIWG TEST SUITE TO INCLUDE DISTRIBUTED SYSTEM TESTS
- DEVELOP A SET OF DYNAMIC ANALYZERS

HUGHES

OVERVIEW

- THE SYSTEM DEVELOPMENT PROCESS
- REAL-TIME DESIGN AND SCHEDULING THEORY
- VALIDATING LARGE, DISTRIBUTED, REAL-TIME SYSTEMS
- IMPROVING THE DEVELOPMENT PROCESS
- → SUMMARY

SUMMARY

HUGHES

- ART INCORPORATES SEVERAL METHODOLOGIES IN SUPPORT OF LARGE, DISTRIBUTED, REAL-TIME SYSTEMS
- A MODERN DEVELOPMENT APPROACH MUST SUPPORT PROTOTYPING, INCREMENTAL, ETC., STRATEGIES
- CURRENT OOA/OOD DO NOT ADEQUATELY ADDRESS LARGE, DISTRIBUTED SYSTEMS
- RTSA AND OOA/OOD METHODS ARE COMPLEMENTARY ANALYSIS AND DESIGN TOOLS FOR LARGE REAL-TIME SYSTEMS
- DEVELOPMENT EFFORTS SHOULD BE CONCENTRATED ON
  - IPC MECHANISMS
  - DISTRIBUTED DATABASES
  - VALIDATION TOOLS
  - EXPANDING EXISTING CASE TOOLS
  - RECOMMENDING OOA/OOD APPROACH
  - EXPANDING EXISTING SCHEDULING MECHANISMS

REFERENCES (1)

ATK88 Atkinson, C. et al., Ada for Distributed Systems, Cambridge University Press, Cambridge, England ,1988.

BEN82 Ben-Ari, M., Principles of Concurrent Programming, Prentice-Hall International, Inc., Englewood Cliffs, NJ, 1982.

CHE76 Chen, P., The Entity-Relationship Model — Toward a Unified View of Data, ACM Trans. on Database Systems, Volume 1, Number 1, 1976.

GOM84 Gomaa, H., A Software Design Method for Real-Time Systems, Comm. ACM, Volume 27, Number 9, September, 1984.

HAC92 ART Guidebook – Volume I: Development Process and Methodology; Volume II: Case Studies and Exercises, Hughes Aircraft Company, October 1992.

HAT88 Hatley, D.J. and Pirbhai, I.A., Strategies for Real-Time System Specification, Dorset House Publishing, New York, 1988.

JHA89 Jha, R. et al., Ada Program Partitioning Language: A Notation for Distributing Ada Programs, IEEE Transactions on Software Engineering, Volume 15, Number 3, March 1989.

LEV90 Levi, S-T. and Agrawala, A.K., Real-Time System Design, McGraw-Hill, New York, 1990.

MEL86 Mellor, S.J. and Ward, P.T., Structured Development for Real-Time Systems, Volume 3: Implementation Modeling Techniques, Yourdon Press, New York, NY, 1986.

REFERENCES (2)

NIE88 Nielsen, K.W. and Shumate, K., Designing Large Real-Time Systems with Ada, McGraw-Hill, New York, 1988.

NIE90 Nielsen, K.W., Ada in Distributed Real-Time Systems, McGraw-Hill, New York, 1990.

NIE92 Nielsen, K.W., Object-Oriented Design with Ada: Maximizing Reusability for Real-Time Systems, Bantam Books, New York, 1992.

SHA90 Sha, L. and Goodenough, J.B., Real-Time Scheduling Theory and Ada, Computer, July 1987.

SHU88a Shumate, K., Layered Virtual Machines/Object-Oriented Design (LVM/OOD), in Proceedings of the Fifth Washington Ada Symposium, ACM, June 27-30, 1988, Washington, DC.

SHU88b Shumate, K., Understanding Concurrency in Ada, McGraw-Hill, New York, 1988.

SHU92 Shumate, K. and Keller, M., Software Specification and Design: A Disciplined Approach fo Real-Time Systems, John Wiley, New York, 1992.

VOL89 Volz, R.A. et al., Translation and Execution of Distributed Ada Programs: Is It Still Ada? IEEE Transactions on Software Engineering, Volume 15, Number 3, March 1989.

WAR85a Ward, P.T. and Mellor, S.J., Structured Development for Real-Time Systems, Volume 1: Introduction & Tools, Yourdon Press, New York, NY, 1985.

WAR85b Ward, P.T. and Mellor, S.J., Structured Development for Real-Time Systems, Volume 2: Essential Modeling Techniques, Yourdon Press, New York, NY, 1985.

Carnegie Mellon University
Software Engineering Institute

Dependable Real-Time Software

L. Sha, R. Rajkumar & J. Lehoczky, 3/11/93

**Software Engineering Institute
Carnegie Mellon University
Pittsburgh, PA 15213**

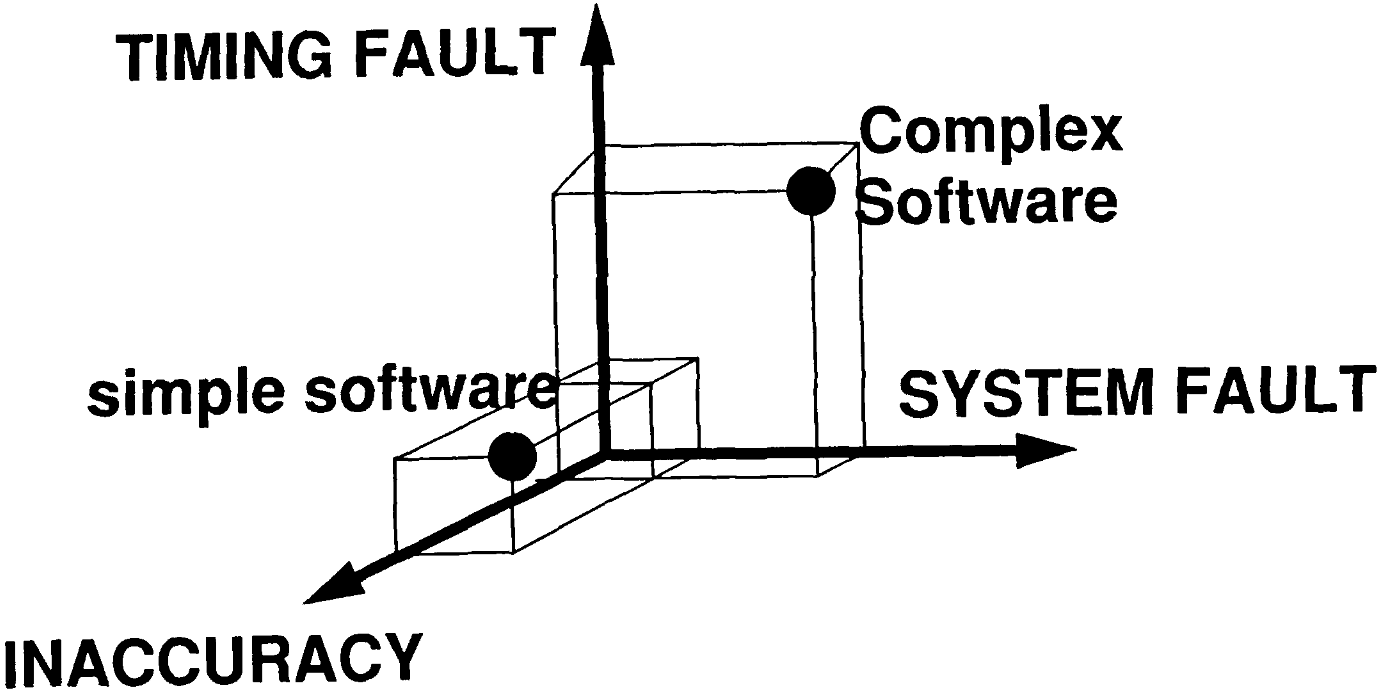
Sponsored by the U.S. Department of Defense

Dependable Real-Time Software

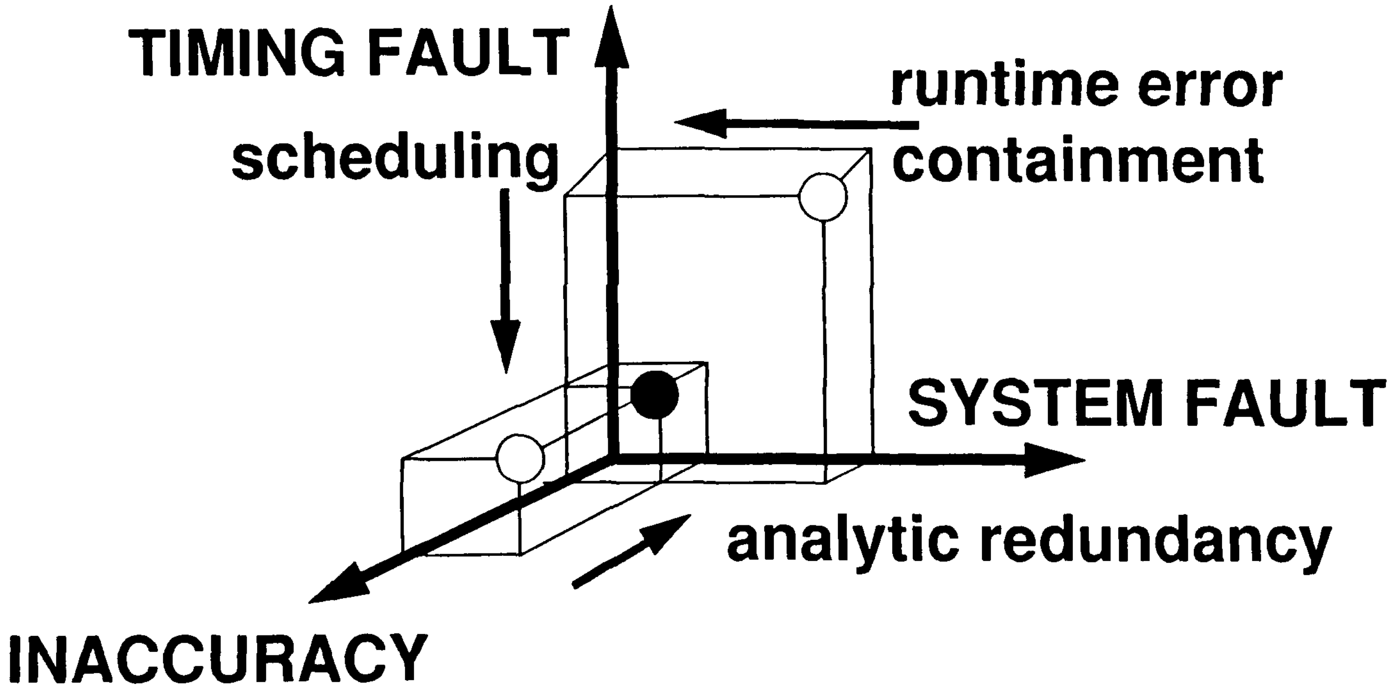
The need

- Predictability and dependability are critical in
  - DoD's surveillance systems, C^3 I systems and weapon systems.
  - civilian systems including telecommunications, air traffic control, automated factories.
- A unified analytical approach is highly desirable.

Fault Model


TIMING FAULT
Complex Software
simple software
SYSTEM FAULT
INACCURACY

Fault Management



Managing Time

To transform the development of real-time system from a trial and error process to an engineering process,

we have developed and transitioned the generalized Rate Monotonic Scheduling (GRMS) theory and associated software engineering approach.

Over the years, we have successfully extended the theory from uni-processor applications to wide area fiber network applications where each distributed scheduler has only delayed and partial state information[Sha & Sathaye, RTSS92].

Managing Time

It has been used successful in very large scale real-time systems such as the data management system of Space Station Freedom and the acoustic subsystem of BSY-2.

RMA is also adopted SPC's ADARTS used for F22 development, selected by the European Space Agency's contract for its on board OS and the Euro-Fighter program.

It is supported by Ada 9x, POSIX.4 and IEEE Futurebus+.

Managing Time

DoD’s 1991 Software Technology Strategy refers to generalized RMS as a “major payoff” of DoD supported R&D. It states that

“System designs can use this theory to predict wether task deadline can be met long before the costly implementation phase of a project begins.

It also eases the process of making modifications to application software”

Managing Time

The Acting Deputy Administrator of NASA, Aaron Cohen, said in his 1992 speech “Charting The Future”,

“Through the development of rate monotonic scheduling, we now have a system that will allow (space station) Freedom’s computers

to budget their time, to choose between a variety of tasks, and to decide not only which one to do first but how much time to spend in the process.”

Managing Time

The National Research Council cited the development of generalized RMS in the selected accomplishment section of its 1992 Report, A Broader Agenda for Computer Science and Engineering. It states

“The traditional method of scheduling concurrent task is to lay out a timeline manually... a change can undo an entire timeline...

This algorithm also enables the convenient accommodation of changes... and is starting to come into use in some real-time aerospace applications.”

Dependable Real-Time Software

It is important to build on the success to unify approach for

- **real-time computing**
- **application level fault tolerance**
- **system level fault tolerance**

and to provide a comprehensive approach to develop advanced prototypical systems and to upgrade the existing systems.

Real-Time Scheduling

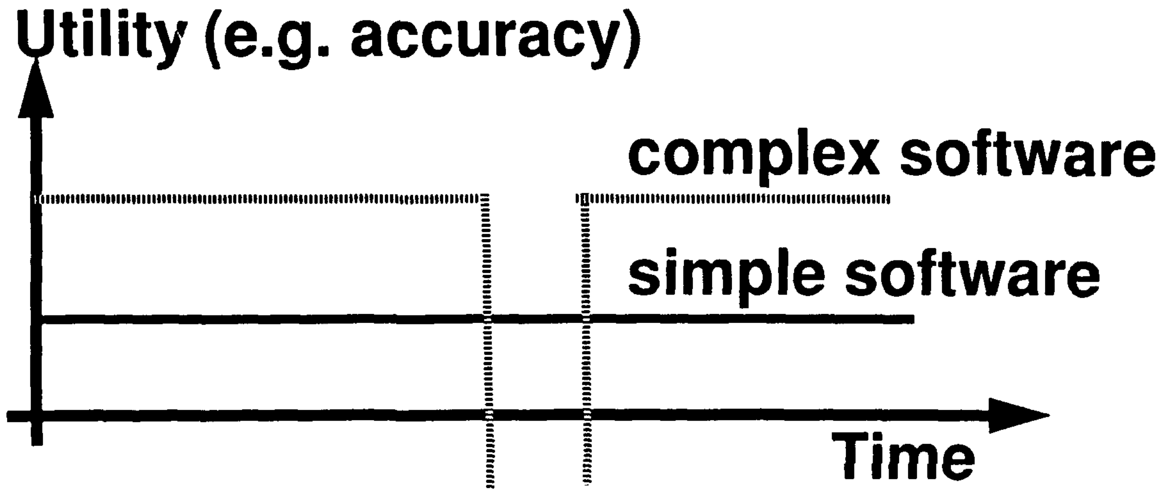
Generalized RMS will be used

- **to schedule the simple software used for analytical redundancy,**
- **to schedule and monitor the complex application software,**
- **to ensure a consistent and timely view of the system state by masking, detecting and recovering from resource failures.**

Application-Level Fault Tolerance

The need

- **A major cause of system faults is complex software.**



The New Paradigm

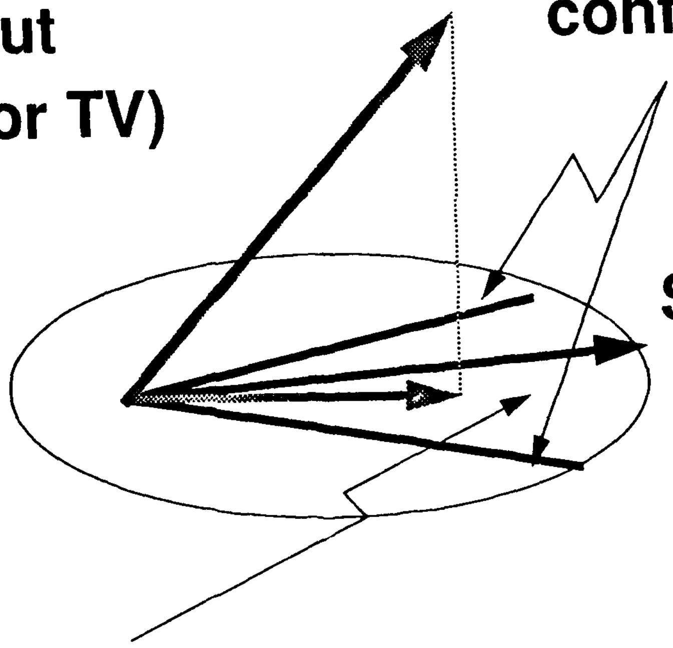
The solution is to let simplicity control complexity.

In the old paradigm, simple alternatives are passive standby. Our method is to turn the old paradigm upside down.

No one can predicate the unknown faults in complex software.

But we can use a simple and defect free application kernel to provide the baseline performance and to control the overall system behavior.

A Conceptual Model



Complex software output (Color TV)

Simple software confidence interval

Simple software output (B&W TV)

Complex software projection

Important Properties of Analytical Redundancy

Simple software computes the baseline answer and a set of constraints, called confidence assertions.

Complex software must observe the confidence assertions produced by the simple software.

Complex software can add functionalities or improve performance even with residual errors.

Guaranteed simple software functional performance, despite timing or logical failures of complex software.

System Fault-Tolerance

In a distributed real-time system, hardware and software resource failures may occur at any time.

Failures must be detected in consistent and timely fashion, and recovery must also be coordinated.

Timely detection and recovery today requires lock-step execution and comparison. Or it cannot be done in real-time.

It is well-recognized that software for lock-step execution is brittle because it is hard to develop, modify and maintain.

Consistency of Global State

The consistency of global state requires only determinism in I/O transactions, which map (analytic) redundant data to unique results.

Cyclical executives maintain determinism by unnecessarily keeping execution images of processors identical and in lock-step.

GRMS analytically guarantees determinism in I/O transactions, and allows for different tasks to execute on different processors and communication channels.

System Fault-Tolerance: Technical Approach

Use RMA technology to ensure the correct timing of:

- **voting and redundant routing for *fault masking*.**
- **distributed membership protocols for *failure detection* and to ensure a consistent view of system state.**
- **checkpoint and restart protocols for *fault recovery*.**

Implementation

Dependability specification

- Specify reliability requirements for hardware resources.
- Specify temporal, reliability and comparison requirements for processes.

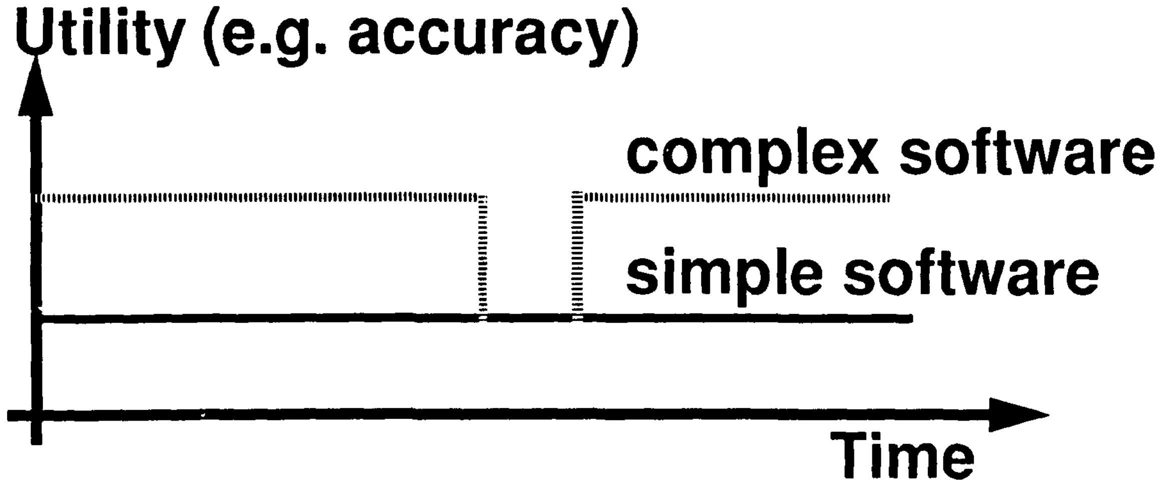
Programming and Run-Time Support

- Extensions to run-time system in the form of kernel/library support.
- Programmer-friendly application interface with well-defined timing semantics.

A Strategic Implication

DoD's new strategy is to build prototypical systems but limit the productions. Lack of use increases the probability of having residual system errors.

Guaranteed baseline performance despite hardware and software failure mitigates technology insertion risk.



A Vision for The Future

We believe in the power of analytical methods for real-time mission critical computing.

We recognize the critical importance of obtaining supports from open standards and use them in the era of tight budgets.

We envision that in the post cold war era the future of real-time mission critical computing lies in dual use.

Our Plan for The Future

Developing the theory, its associated standard computing architecture and customizable components for data acquisition, control and communication to reduce the cost and risk of developing and maintaining mission critical systems.

Adapting the architecture and its customizable components for industrial applications where predictability and reliability is a serious concern, such as the operation of chemical plants, air traffic control and high speed mass transit systems.

Options for Validating Real-Time Systems

Jonathan D. Wood
Institute for Defense Analyses
March 17, 1993

Question: How to collect data in real-time systems without disturbing system timing?

- General considerations

- Transporting a real-time system to new hardware changes system timing, but upgrading to the latest, fastest, cheapest hardware every few years is compelling.

- Software Engineering experience has been that testing is 50% of the total lifecycle effort.

- Not all real-time systems need the same level of effort in data collection

- Formal methods are not yet an option for real-time systems

- We <u>will</u> make mistakes

- Effecting time more a problem than effecting space

• Options for collecting data

- Hardware monitoring
  - — Doesn't effect timing
  - — Expensive and involved
  - — Doesn't survive a hardware port
- Temporary software monitoring
  - — Problem with temporary data collection is that its presence is intrusive
  - — Changes include different synchronization points, different timing
- Permanent software monitoring

- **Permanent software data collection: leaving the real-time data collection subsystem in the system.**

- Advantages of leaving it in are: no change in synchronization points, no change in timing, capture of all occurances of system misbehavior.

- If there is no way of producing a particular error on demand, then data must always be collected so that whenever the system misbehaves, detailed data sufficient to deduce what happened is available.

- Claim: a good real-time design has permanent data collection designed into the system from the outset

- Tools to rapidly build data collection subsystems would be useful.

Recorder

- Requirements for measurements of RTS (Remote Temperature Sensor) system timing changed
- Previously measured only throughput
- Needed to measure end-to-end duration of control packet transit time
- Wanted much more information about times of events and relationships between events

Recorder

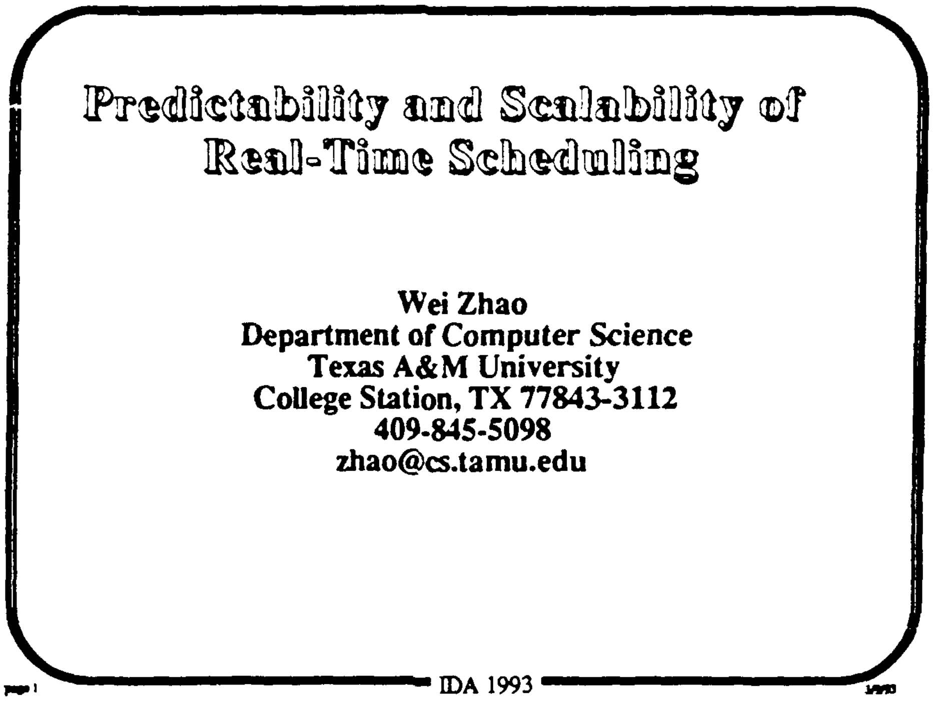
- Previous method of “write event to screen” unsatisfactory
- Writes to screen became scrambled with no screen mutual exclusion, especially on multiprocessor
- Mutual exclusion of screen changes synchronization points
- Getting more information about system under test required more than writing to output device

Recorder

- **Recorder Features**
- Introduces no new synchronization points
- Is time-efficient and space-efficient
- Doesn't write to screen
- Doesn't require coordination between processes during data collection

Recorder

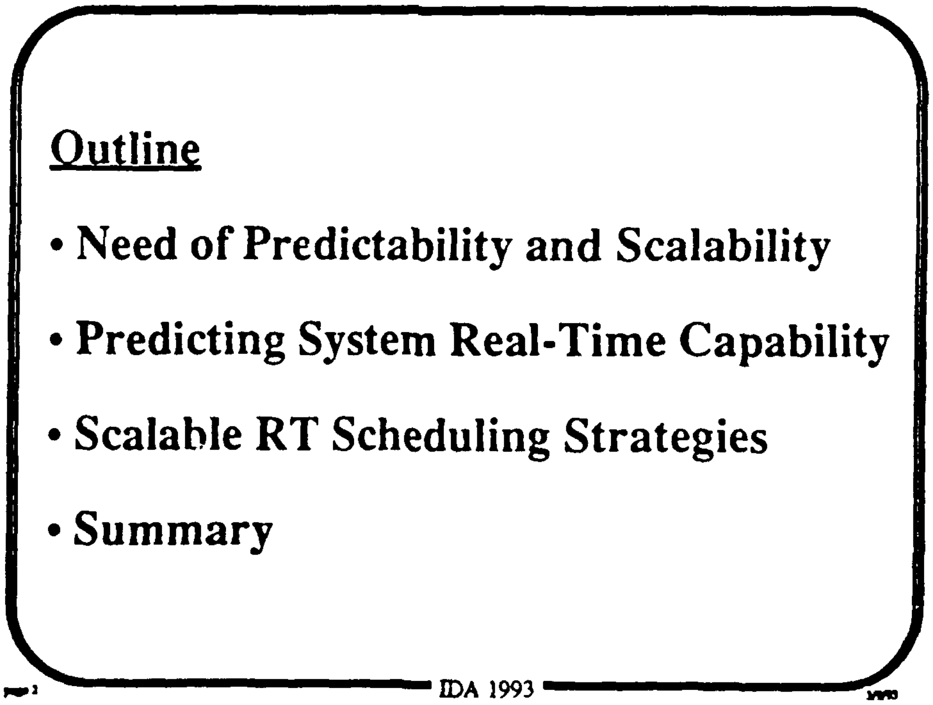
- **Recorder Interface**
- Ada generic package specification
- Can record records, integers, enumeration types, etc
- Provides procedures to log (and timestamp) events, sort events into cronological order, and generate statistices about events.
- Log procedures are inlined to avoid calling overhead



Predictability and Scalability of Real-Time Scheduling

Wei Zhao
Department of Computer Science
Texas A&M University
College Station, TX 77843-3112
409-845-5098
zhao@cs.tamu.edu

page 1 IDA 1993 [illegible]



Outline

- Need of Predictability and Scalability
- Predicting System Real-Time Capability
- Scalable RT Scheduling Strategies
- Summary

page 2 IDA 1993 [illegible]

Need of Predictability and Scalability

IDA 1993

Trend of Real-Time Computing

- Limited budget
- Advancing with technology

$$\frac{\text{Design Cost}}{\text{Production Cost}} \nearrow$$

Making smart (scientifically justifiable) design decisions

System performance must be predicable.

IDA 1993

Trend of Real-Time Computing (cont.)

- Massive parallel processing
- High speed networking

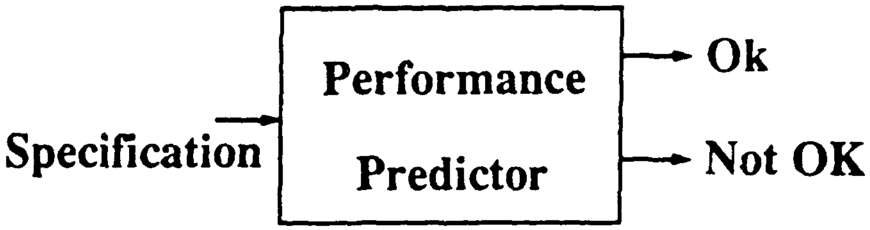
$$\frac{\text{Resource Management Overhead}}{\text{System Throughput}} \nearrow$$

Resource management must be scalable.

Predicating System Real-Time Capability

- Requirements
- Utilization based predications
- Successes and problems

Requirements of Performance Predictor



Specification

Performance Predictor

Ok

Not OK

page 7

IDA 1993

- Predicating system safety margin is the most essential.

  ==> Predication must be stable

- Predication should not depend on detailed system specification.

  ==> Utilization based predication

page 8

IDA 1993

Utilization based predication:

- Worst Case Achievable Utilization

  If the demand is less than WCBU, real-time constraints are met.

  This is a measure of the worst case.

Progress in WCAU

- C Liu et al (1973)

  RMS for a single cpu system,

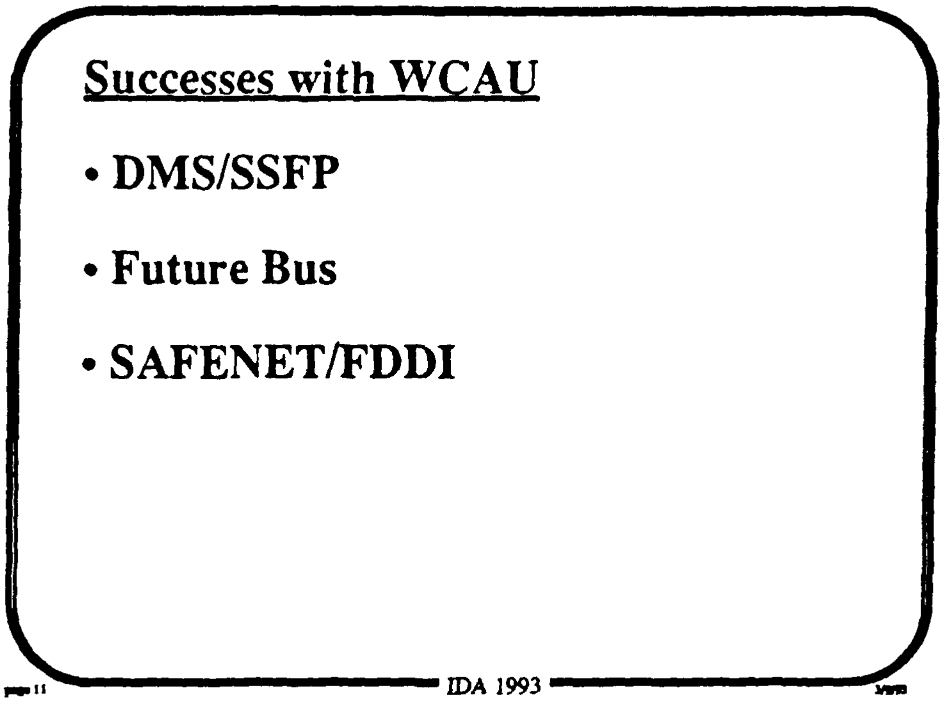
  WCAU = 69%.

- CMU Research Group (1980s)

  WCAU for various environments.

- TAMU Research Group (1992)

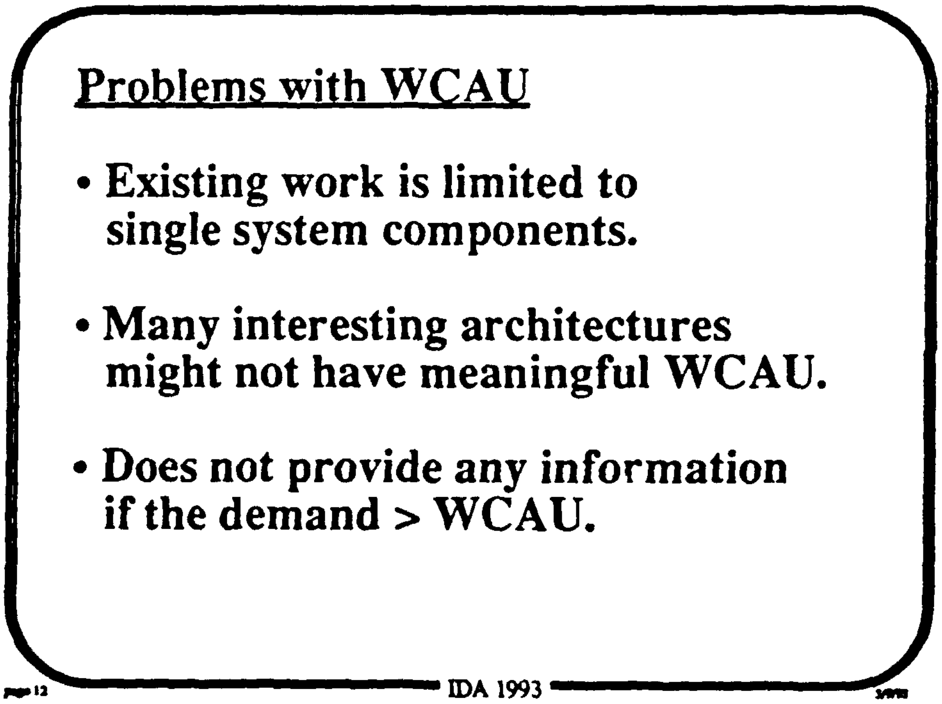
  WCAU(FDDI) = 33%.



Successes with WCAU

- DMS/SSFP
- Future Bus
- SAFENET/FDDI

page 11 IDA 1993



Problems with WCAU

- Existing work is limited to single system components.
- Many interesting architectures might not have meaningful WCAU.
- Does not provide any information if the demand > WCAU.

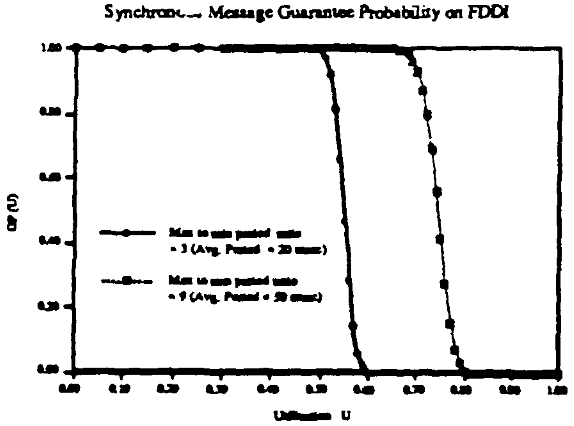
page 12 IDA 1993

Another utilization based predication:

Guarantee Probability

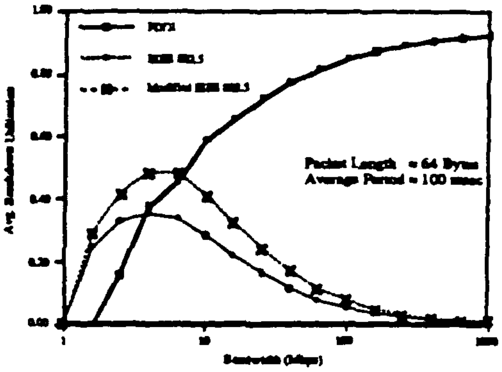
- Definition
  GP(U) =
  Prob(A set of requests
  are guaranteed | demand = U)
- A measure of both average and worst cases.
- A generalized notion of WCAU:
  WCAU = max(U | GP(U) = 1).

Guarantee Probability for FDDI



Number of stations N = 100.
Minimum Period = 10 msec.

The 3rd utilization based predication:
Average Breakdown Utilization

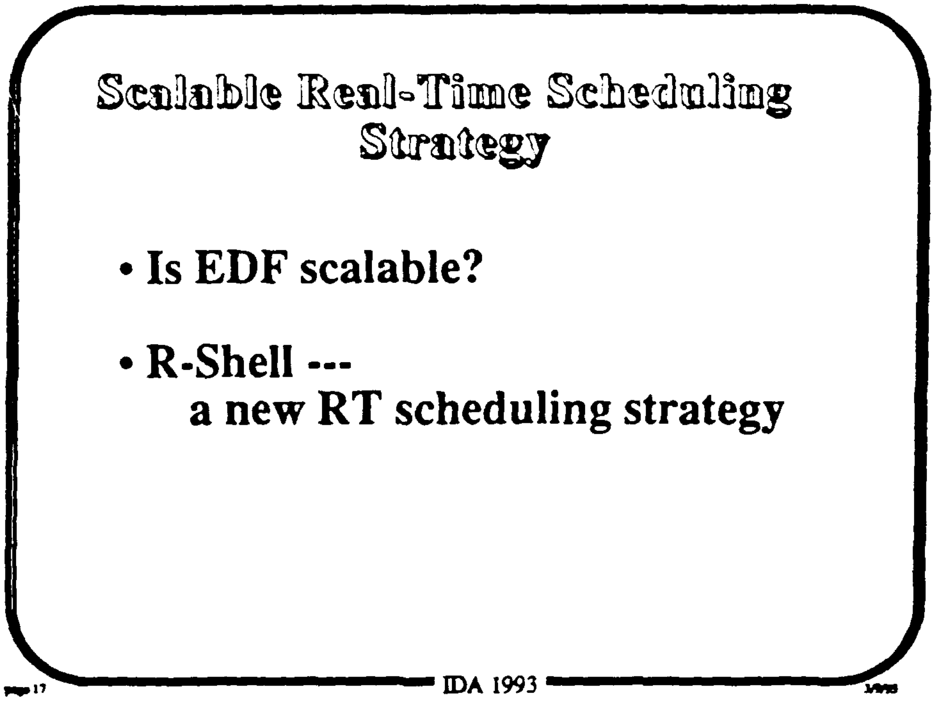


page 15 IDA 1993

Utilization based predication helps to

- make design decisions
- establish management confidence
- reduce testing and integration cost

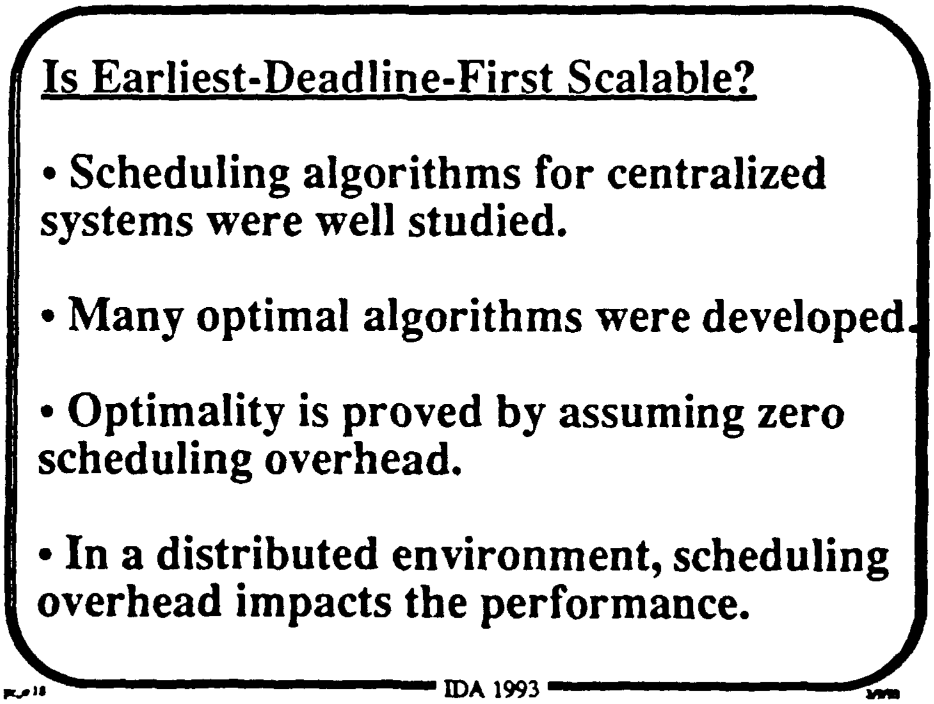
page 16 IDA 1993



Scalable Real-Time Scheduling Strategy

- Is EDF scalable?
- R-Shell ---
  a new RT scheduling strategy

page 17 IDA 1993



Is Earliest-Deadline-First Scalable?

- Scheduling algorithms for centralized systems were well studied.
- Many optimal algorithms were developed.
- Optimality is proved by assuming zero scheduling overhead.
- In a distributed environment, scheduling overhead impacts the performance.

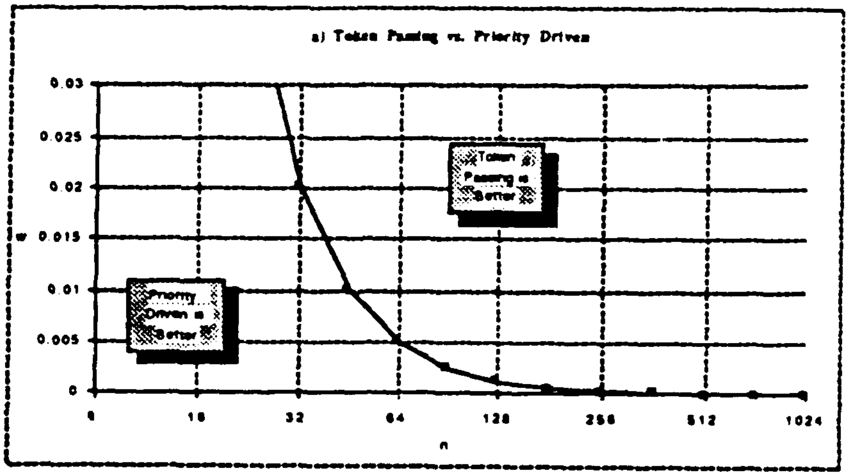
page 18 IDA 1993

An Example: scheduling real-time messages in a token ring network

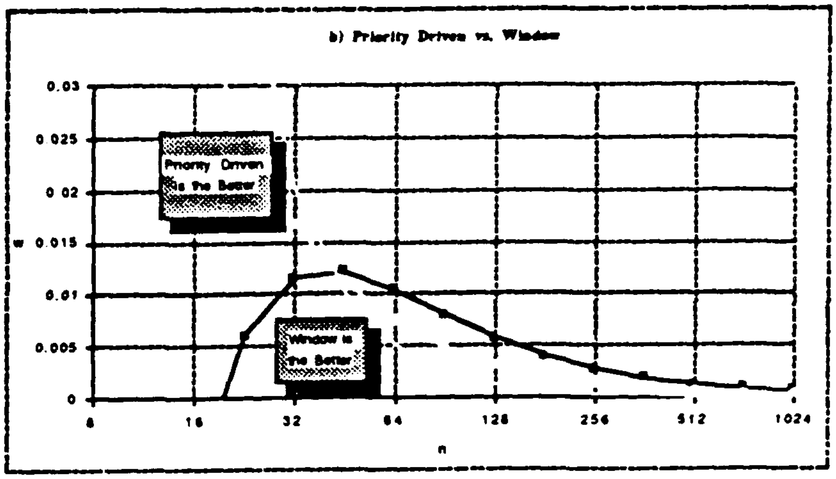
Three protocols to to studied:

- Simple token passing protocol
  --- Non EDF
- Priority driven protocol
  --- Approximate EDF
- Window protocol
  --- Exact EDF

page 19 IDA 1993

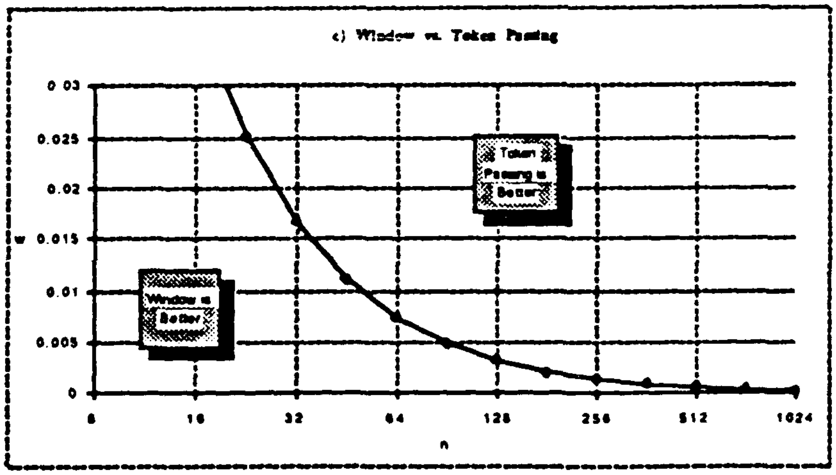


IDA 1993



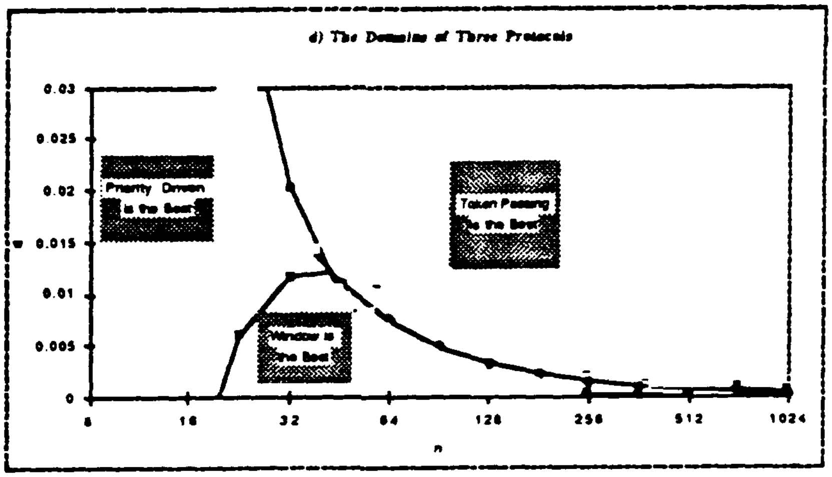
b) Priority Driven vs. Window

Page 21 IDA 1993 [illegible]



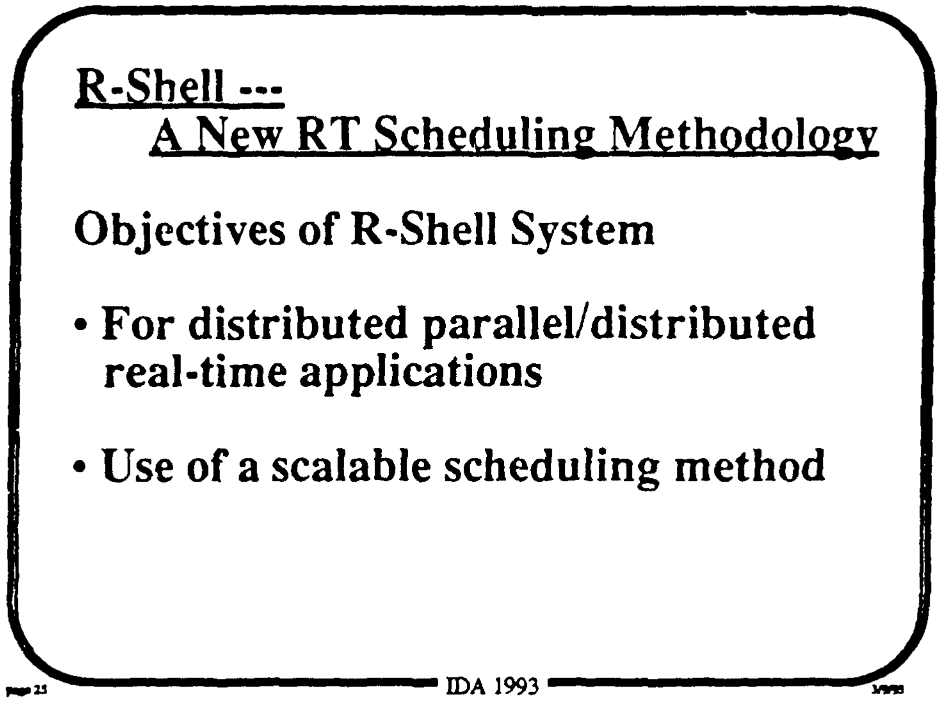
c) Window vs. Token Passing

Page 22 IDA 1993 [illegible]



In a parallel/distributed system,

$$Performance = \frac{\text{Schedduling Policy}}{\text{Scheduling Overhead}}$$

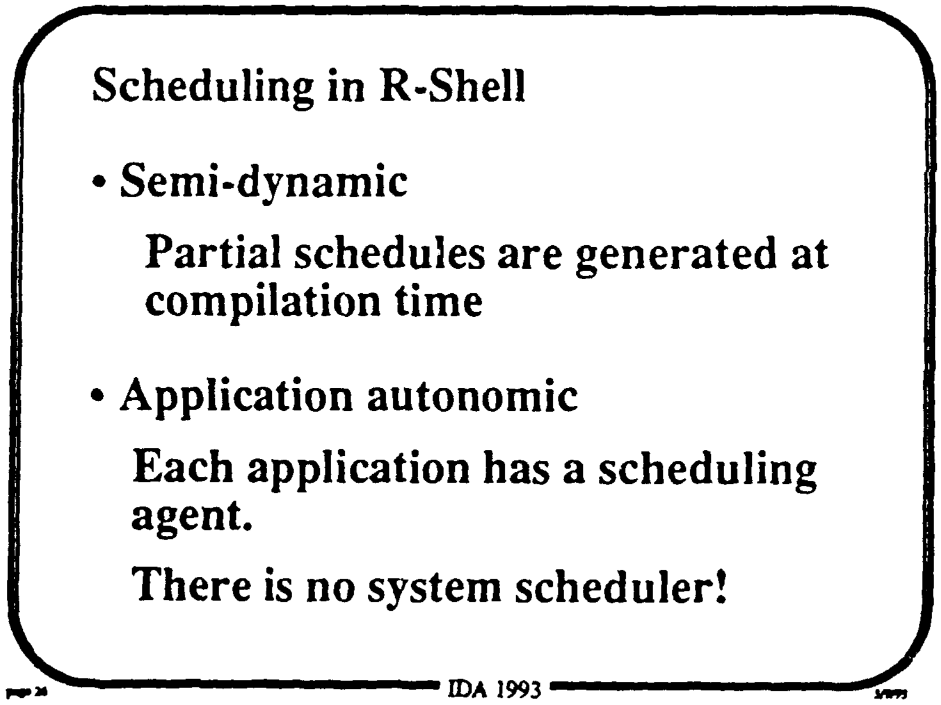


R-Shell ---
A New RT Scheduling Methodology

Objectives of R-Shell System

- For distributed parallel/distributed real-time applications
- Use of a scalable scheduling method

page 25 IDA 1993 [illegible]



Scheduling in R-Shell

- Semi-dynamic

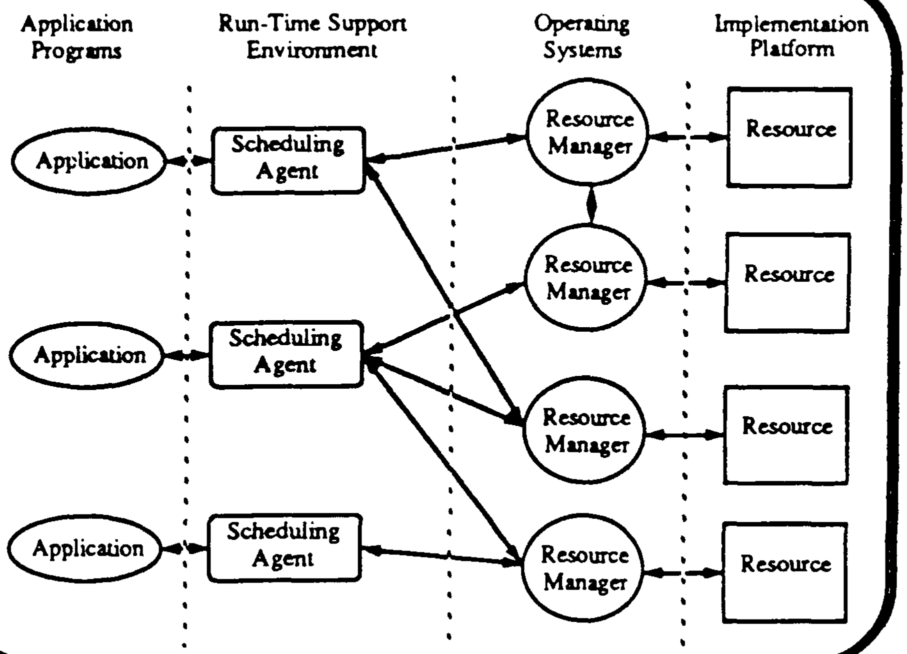
  Partial schedules are generated at compilation time

- Application autonomic

  Each application has a scheduling agent.

  There is no system scheduler!

page 26 IDA 1993 [illegible]



Summary

- Real-time scheduling in parallel/distributed system is challenging
- Predictability and scalability are two key issues
- Further systematic exploration should result in cost-effective design methodology for new generation RT computing systems

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE July 1993	3. REPORT TYPE AND DATES COVERED Final

4. TITLE AND SUBTITLE
Proceedings of the Workshop on Large, Distributed, Parallel Architecture, Real-Time Systems

5. FUNDING NUMBERS
MDA 903 89 C 0003
Task T-R2-597.2

6. AUTHOR(S)
Norman R. Howes, Dennis W. Fife, Jonathan D. Wood

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Institute for Defense Analyses (IDA)
1801 N. Beauregard St.
Alexandria, VA 22311-1772

8. PERFORMING ORGANIZATION REPORT NUMBER
IDA Document D-1425

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Ballistic Missile Defense Office
The Pentagon, Room 1E149
Washington, DC 20301-7100

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; unlimited distribution: 8 June 1994.

12b. DISTRIBUTION CODE
2A

13. ABSTRACT (Maximum 200 words)

The workshop on Large, Distributed, Parallel Architecture, Real-Time Systems was sponsored by the Ballistic Missile Defense Office (BMDO) and the NASA Ames Research Center and hosted at IDA in March 1993. The purpose of the workshop was to obtain expert opinions on the following questions: (1) What is the best design methodology for this class of systems? (2) What is the proper relationship between design theory and scheduling theory? (3) What is the best method for validating this class of systems? and (4) What are the most promising areas where resources might be applied for near-term benefits? Twenty-three experts from academia, government and industry were invited to attend of which seventeen accepted. In total, there were twenty-three participants including sponsors and IDA research staff. The invitees contributed position papers in advance of the workshop and presented talks from transparencies. These position papers and transparencies comprise the contents of the proceedings. The informal discussions that took place at the workshop are summarized in the introductory material by the IDA research staff. The advice, opinions and methods of the participating experts are intended to help BMDO and NASA in the development of their respective software technology planning.

14. SUBJECT TERMS
Real-Time; Parallel Real-Time; Distributed Real-Time; Real-Time Scheduling; Real-Time Design Methods.

15. NUMBER OF PAGES
506

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT Unclassified	18. SECURITY CLASSIFICATION OF THIS PAGE Unclassified	19. SECURITY CLASSIFICATION OF ABSTRACT Unclassified	20. LIMITATION OF ABSTRACT SAR