

Designing Distributed, Real-Time Systems

Kevin L. Mills

INFT 796 SUMMER 1993
DIRECTED READINGS IN SOFTWARE ENGINEERING
WITH DR. H. GOMAA
GEORGE MASON UNIVERSITY

Designing Distributed, Real-Time Systems

Kevin L. Mills

August 25, 1993

In a 1987 article considering future prospects for increasing the productivity of software developers, Frederick P. Brooks identified inherent and arbitrary complexity as two fundamental properties of software that limit the productivity gains software developers can expect to achieve. Dr. Brooks based his thesis on his experiences leading the design and development of the original IBM/360 operating system, where he first encountered the complexity of software systems, and on the two decades since, during which software engineering research has improved productivity marginally by addressing those aspects of software design and development that Dr. Brooks views as accidents. In the years since Dr. Brooks' sage article appeared, software system design and development has continued to increase in complexity as computers are applied to more problems, problems that increasingly involve real-time requirements and distributed computing. Complexity, both inherent and arbitrary, remains, then, an essential problem faced by designers of software systems, and particularly by designers of distributed, real-time systems.

The present paper investigates the nature of complexity as pertaining to design of distributed, real-time systems. Three main questions are considered. First, what problems face designers of distributed, real-time systems? Answering this question reveals the essential complexity inherent in the software for such systems. Second, what methods can designers use to address the problems they face? Some of the methods discussed are currently used routinely by designers, while

others remain the subject of research. The present paper evaluates design methods against the needs of distributed, real-time system designers. Finally, the paper considers how software design environments might improve a designer's ability to manage the complexities of designing distributed, real-time systems. To address these questions, seven sections follow this introduction.

Section II, *The Design Problem*, begins by examining the general nature of design: its definition, its purpose, and associated activities. The concept of design methods is introduced as an essential tool to assist designers. The section then delves into specific goals that must be achieved by designers of distributed, real-time systems. The section closes with a discussion of the special considerations faced when a real-time system is also distributed.

Section III, *Some Design Approaches*, provides a designer's-eye view of the current practice of real-time system design. The section begins with a discussion of the question of schedulability. The major approach to designing hard-real-time systems (HRTs) over the past three decades revolves around a fixed schedule of module executions, computed off-line, coupled with a cyclic executive that enforces the schedule. In general, this approach results in deterministic software that meets all real-time requirements, but also in a software system that is difficult to understand and maintain. More recent approaches treat real-time software as software systems first and real-time systems second. This means that these approaches are used to design software that, while understandable and maintainable, is concurrent, and thus operates non-deterministically. Such non-determinism traditionally calls into question the schedulability of concurrent designs; however, concurrent design approaches are growing in popularity due to a new scheduling theory called rate monotonic analysis (RMA).

Depending on which view a designer takes on the question of schedulability, different design approaches might prove necessary. This paper examines two general design approaches, deterministic and concurrent, and considers some examples of each approach.

Having considered the problems faced by designers and then having examined some design approaches, the paper recapitulates, in section IV, a set of open issues in the design of distributed, real-time systems. The issues identified represent the hard problems that designers must solve, but for which no routine solution is available.

Section V, *Formal Methods for Designers*, reviews formal models and methods that various researchers believe might address the open issues identified in section IV. Most of the formal models and methods discussed are supported by automated tools. For each method, the basic notation, model and properties are described, some specific examples are discussed, and, where applicable, a few representative automated tools are identified. The discussion includes a summary of the strengths and weaknesses of each method.

In some cases, the formal models and methods reviewed in section V comprise a foundation for languages that can be used to describe designs and then to implement prototypes of those designs. Section VI, *Languages for Designers*, considers several design languages that embody formal models and methods. Included in the discussion of each language are: 1) the basic notations, semantic models and properties, 2) some representative implementations, and 3) the strengths and weaknesses.

Section VII, *Design Environments*, synthesizes the concepts investigated in previous sections of the paper. Synthesis is achieved by envisioning a design environment that might enable the designer of distributed, real-time systems to develop and describe understandable designs that are functionally correct

and that meet specified performance requirements. The desirable traits of such a design environment are sketched, then a few example design environments are described and evaluated against the set of desired traits.

A concluding section (VIII) provides a summary of the ideas advanced in the paper. Designers of software systems are challenged by an inherent complexity; and the most complex software known today is embedded in real-time systems. In the future, as real-time system components become distributed, the complexity of such software will jump. While approaches exist today to deal with the design of real-time systems, some significant open issues remain. Additional issues arise when real-time systems are also distributed systems. Researchers are investigating formal methods and models, and related languages, for addressing many of the problems faced by designers of real-time and distributed systems. In some cases, researchers propose design environments to assist the designer through an integrated set of tools. This paper attempts to identify the desirable traits of an environment for designing distributed, real-time systems, to show that the current state of research regarding software design lacks maturity, and to identify some of the more promising avenues for continued work.

II. The Design Problem

The design problem is similar in nature to the problem faced by the author of this paper as he sits at a keyboard and gazes upon a white sheet of paper. The author knows in the main what to say but he wonders just how best to say it. This problem is fundamentally different from the problem of a natural scientist. A scientist examines the world around us in an effort to discern cause and effect relationships and to describe those relationships in the form of mathematical equations and

scientific laws that enable us to predict the outcome of various physical situations. In short, a natural scientist is concerned with what is, and why. A designer, on the other hand, is concerned with what ought to be, and how.

This essential difference between natural science and design led Herbert Simon to include design within the category of disciplines that he dubbed the sciences of the artificial. [SIMO81] According to Simon, "[d]esign...is concerned with how things ought to be, with devising artifacts to attain goals." [SIMO81, p. 133] Four other, similar, views of design were reported by Peter Freeman [FREE80] in a survey he conducted: 1) design is an imaginative jump from present facts to future possibilities, 2) design is finding the right components of a structure, 3) design is decision-making in the face of uncertainty with high penalties for error, and 4) design is simulating, iteratively, a proposed solution until confident about the outcome. Freeman goes on to suggest that design has three purposes.

One purpose of design is to discover the structure of a problem. Within the realm of software this purpose might be fulfilled by reviewing the informal software requirements specification and then by analyzing the requirements using some systematic method. A second purpose of design is to create an outline, or architecture, of a solution for a problem. For software design, this purpose might be met by describing a set of software components and the relationships between them in enough detail that further design and then coding can be performed on each component. A third purpose of design is to evaluate the results of proposed architectures against the stated goals (i.e., the requirements). For software design, this purpose is often handled poorly. Typically, evaluation is delayed until system testing. Design flaws discovered during system tests can be quite costly to repair. A more modern

approach employs rapid prototyping to validate the informal requirements; however, prototypes often encode a de facto solution to the requirements and thus usurp a designer's ability to propose and evaluate various solutions.

To meet his purposes a designer usually engages in a number of intellectual activities. [FREE80] One such design activity might be called *operationalization*. Operationalization entails improving the informal requirements so that ambiguities are removed, inconsistencies are reconciled, and incompleteness is removed. This is a necessary part of the designer's job because later design activities depend upon the system requirements. Another design activity involves *abstraction*. Here the designer generalizes about particular properties of the problem or of a possible solution; certain details are set aside at critical moments so that the designer can concentrate on a specific issue. Associated with abstraction is *elaboration*. A designer employs elaboration to move down a hierarchy of levels of abstraction so that essential details can be provided at an appropriate time. Probably the most important intellectual act during design is *verification*. A designer must verify that a proposed solution meets the requirements, any imposed standards, and any extant constraints. A designer must also be able to verify the performance characteristics of a proposed solution.

The essence of design, as embodied by the four intellectual activities of operationalization, abstraction, elaboration, and verification, is *decision-making*. Unfortunately, the record reveals that designers do not always make sound decisions.

Experience with large software systems shows that over half of the defects found after product release are traceable to errors in early product design. Furthermore, more than half the software life-cycle costs involve detecting and correcting design flaws. [BERE84, p. 4]

To ameliorate these problems researchers have focused on the development of design methods. Several design methods for distributed and real-time systems are discussed in section III of this paper, but for now consider, in general, how a design method can help. A design method specifies: 1) what decisions a designer must make, 2) how those decisions should be made, and 3) in what order they should be made. [FREE80] A design method, then, should provide the intellectual roadmap that enables a designer to refine requirements successfully, to apply abstraction and elaboration correctly, and to achieve design verification. Design methods aim to improve the skills of software designers so that the designs produced by designers using a given method achieve a reasonable quality on a repeatable basis.

To this point in the paper the reader should have gained a general understanding of design, of the purposes of design, of the intellectual activities involved in design, and of the way in which design methods might aid a designer. From here, the discussion becomes more specific to software design, and particularly to design of distributed, real-time software.

A. Design Goals For Distributed, Real-Time Software

The goals for designers of distributed, real-time software build upon the goals for designers of general software systems. Before considering specific design goals, a short discussion to distinguish distributed, real-time software from general software may prove helpful. Software, generally, is designed and implemented to fulfill a set of functional requirements and non-functional requirements. Functional requirements express the necessary logical characteristics of a correct solution. Non-functional requirements describe other operational constraints, such as performance, reliability, and specific target hardware. For real-time systems, the non-functional

requirements take on an added importance. For so-called soft real-time (SRT) systems (sometimes referred to as interactive systems) the performance requirements might indicate a performance target given a specified load on the system; for example, "95% of all transactions will be processed in under five seconds when the system load peaks at 100 transactions per second." The understanding of such requirements is that when system load exceeds the peak, or on five percent of the occasions that the load is at or below peak, system performance may degrade without any real harm. For so-called hard real-time (HRT) systems (sometimes referred to as reactive systems) the performance requirements can form a three-level hierarchy: 1) those that must be met for correct system function, 2) those that are soft (in the sense formerly discussed for SRT systems), and 3) those that have more lenient time constraints (usually called background functions). An example of a HRT requirement might be that "a temperature sensor shall be polled every 100 ms." For such a requirement, a software solution that polled the sensor twice at 101 ms apart would be inadequate. For real-time software, then, the performance requirements take on a functional flavor in that a system that does not meet the stated performance constraints is considered functionally degraded for soft real-time requirements and is considered functionally incorrect for hard real-time requirements.

While real-time requirements complicate software design by giving a functional flavor to some otherwise non-functional requirements, distribution of software functions among several processors introduces another type of complexity. Distribution of software functions ensures that concurrent processing will occur. Concurrency leads to a hidden set of correctness requirements involving inter-process synchronization and communication. The requirements arising from concurrency are seldom mentioned specifically in a software requirements

document but a system will be unable to meet its stated functional and non-functional objectives unless concurrency is properly handled.

Given the foregoing discussion of real-time requirements, distribution and concurrency, the reader may be surprised to learn that designers of distributed, real-time systems aim to achieve the same three general goals as designers of any software: 1) understandability, 2) functional correctness, and 3) performance sufficiency. Surprised or not, the reader should already suspect that meeting these goals will be more difficult for designers of distributed, real-time software than for designers of sequential, non-real-time software. The following paragraphs confirm the reader's suspicions.

To achieve understandability the software designer must meet four sub-goals. First, the designer must ensure complete, consistent, and unambiguous functional requirements. Software requirements documents typically consist mostly of natural language descriptions augmented with some formal specifications that are generally applied unevenly. The designer must seek to improve the rigor of the specification, to fill the gaps, and to resolve contradictions. Without such efforts the designer cannot achieve an understanding of the problem sufficient to propose and evaluate solutions. The remaining sub-goals relate directly to design.

The designer must provide a clear structuring of the system into processes and information hiding modules. Then the designer must specify the behavior of the processes and the functions of the information hiding modules. Finally, the designer must establish traceability between the structure and specification of the design and the software requirements. The result of achieving these sub-goals, is an understandable, but static, design of a software architecture.

Next, the designer must work to ensure the functional correctness of the design at the component level and at the architectural level. At the component level, the designer should specify partial correctness criteria for each sequentially executing path. Such paths typically include the program flow of control (one for each task when the design is concurrent) and the services provided by each information hiding module. In general, the designer should specify preconditions and post-conditions for each design component such that if the preconditions of the component are satisfied on entry to the component, then the post-conditions will hold upon exit from the component. These specifications will enable component designers and coders to understand precisely what their component must achieve, as well as to understand what should be provided to and expected from components with which their components interact. Such specifications can also serve as a foundation for unit and integration testing as the design is implemented.

At the system level, designers of concurrent systems have two concerns regarding functional correctness. One concern involves ensuring the absence of undesirable properties, such as deadlock, livelock, unfairness, failure, and unreachable states, that can occur in concurrent designs. [KARA91, LIU90, LEVI90, XU93] Deadlock occurs when two or more tasks cannot proceed with processing because they are waiting on resources that are held by each other or they are waiting to synchronize at mutually conflicting points. Deadlocks can creep into a design in a variety of ways and can be difficult to detect, to isolate, and to eliminate. Livelock occurs when one or more tasks in a concurrent system continue to cycle but are unable to make any progress. Livelock is a particular problem in distributed systems where normal behaviors may be repeated indefinitely due to an aberrant design. Unfairness occurs when one or more equal priority tasks, among a competing set, are given preferential

access to a resource, or when one or more higher priority tasks consume so much of a resource that an inadequate amount is left for lower priority tasks. Unfairness comes in two forms: hunger and starvation. A task suffers hunger when an insufficient amount of a needed resource is available. A task suffers starvation when none of a needed resource is available. Failure occurs when tasks attempt to interact but find that conditions prevent such interaction or when an unhandled exception occurs within a task. Unreachable states result when a design includes logic for handling conditions or events that cannot occur. Such unreachable states may result from an inadequate design or from poorly understood system requirements.

A second concern of the designer, regarding functional correctness at the system level, is to establish that a concurrent design exhibits certain desirable properties, such as proper synchronization among communicating tasks, mutually exclusive access to shared resources, bounded behavior, and conservation of system resources. [DILL90, MURA84, MURA89, WILL90, ZAVE86] Proper synchronization ensures that tasks obtain the necessary input before executing and that external events are properly ordered by the software. Controlling shared access to system resources prevents corruption of system data. Ensuring bounded behavior prevents queue overflow and the subsequent loss of external or internal events. Verifying conservation of resources ensures that the software does not consume resources that are intended to persist for the duration of the execution.

The final concern of the designer is to meet the performance constraints for the system. [LIEN92, LIU90, NATA92, LEVI90, XU93] An initial complication arises when the timing constraints in the requirements specification are not complete or consistent. So the first concern of the designer is to properly specify the system performance constraints. After a

system's timing requirements are properly understood, the designer's major performance concern, for hard real-time systems, becomes ensuring schedulability of the software design under worst-case assumptions. This involves estimating or prescribing the worst-case execution time of each design component and then establishing that the software will meet all deadlines for periodic processes, will achieve the required response time for aperiodic events, and will maintain stability under transient, peak loads. A "...perplexing aspect of this [time] problem is that most system design and verification techniques are based on abstraction, which ignores implementation details...[but]... timing constraints are derived from the environment and the implementation." [STAN88, p. 14] A subsidiary concern of the designer is to maximize the software performance under typical, sustained loads.

In summary, designers of software are concerned with creating understandable designs that can guide implementation and provide traceability to the requirements specification; however, when the designs include concurrency a number of implicit functional requirements must be addressed. Concurrent designs must be free from deadlock, livelock, failures, unfairness, and unreachable states; at the same time concurrent designs must exhibit proper synchronization and resource sharing among tasks, must exhibit boundedness, and must conserve system resources. Designers of real-time systems must also be concerned about specific performance characteristics: 1) maximal performance under a sustained load for interactive systems and 2) worst-case performance under transient loads for reactive systems. Many of the issues faced by designers of concurrent, real-time systems can only be addressed through a dynamic evaluation of the software design. Unfortunately, such dynamic evaluations often occur only after the system is implemented.

As complicated as concurrent designs can be, concurrent systems are actually a subset of distributed systems. That is, distributed systems are naturally concurrent, but concurrent systems need not be distributed. When a concurrent system is also a distributed system, the software designer must address a special set of issues that can further complicate the design. These special considerations for distributed systems are discussed next.

B. Special Considerations For Distributed Systems

Designers of distributed systems face an extra decision during system structuring -- the allocation of processes and data to nodes. [ROFR92, SUMM89] Distributed system design methods generally provide guidelines to help a designer with these decisions; however, the effect of such decisions on system performance and on implicit functional correctness remain no better addressed than is the case for concurrent designs.

When processes and data are distributed among nodes, further complications arise due to uncertainties regarding inter-node communication. [KLEI85, SHAT84, STAN82, STAN88] An initial complication is selecting a suitable inter-node message-passing paradigm to use. Within concurrent software, asynchronous message-sending (sometimes called loosely-coupled communication) provides a natural model for producer-consumer relationships. Of course, in centralized, concurrent designs the loss of a message is seldom of concern. Should tasks in separate nodes need to communicate, yet remain decoupled, some sort of asynchronous message-passing must be provided between nodes. In such cases, the error properties (discussed below) of the communications path become a grave concern.

When synchronous message-passing between tasks on separate nodes is needed, a decision must be made whether to support synchronization with or without reply, or both. Decisions taken here will dictate the requirements that must be met by the

inter-node communication protocols. Should inter-task rendezvous be needed across nodes, then synchronous message sending with reply will likely be required.

In the event that a client-server relationship exists between tasks on separate nodes, synchronous message-passing with reply might provide a natural means to implement a remote procedure call (RPC) mechanism. Even in this case, the designer must know what semantics the underlying RPC protocol will provide. Some RPC protocols can guarantee "at-least-once" semantics, i.e., a remote call will be executed at least once, but maybe more than once. Other RPC protocols provide "exactly-once" semantics, i.e., a remote call will be executed exactly once. Even with these issues settled, a semantic is needed to interpret exceptions returned from RPCs.

Aside from the many possible paradigms for sending messages via network, the designer of distributed systems must also be concerned with paradigms for receiving messages from a network. When a central system is used to pass messages between tasks, the semantics are provided by the operating system or real-time executive. When a system is distributed around a network, the designer must become involved in the message reception semantics that are needed for a particular design. A receiver might wish to wait for any message arriving at a queue. A receiver might also wish to wait only for some specific message or on a selected set of messages. Perhaps a receiver needs to wait on a set of message queues based on priority. Whatever decisions are made regarding message reception paradigms, a suitable set of protocols must be designed and implemented. The reader should bear in mind that the protocol processing itself constitutes a distributed, concurrent system that may also face hard, real-time requirements.

In addition to selecting paradigms for sending and receiving messages, the designer must determine the level of integration

needed between the mechanisms for external communications, internal events, and external interrupts. When these paradigms are integrated (as they are for example in the Ada language), the designer's task may be significantly eased. On the other hand, achieving the required level of integration may prove impractical, especially when the nodes execute under different operating systems.

Another consideration for the distributed system designer is the need for multi-addressee message passing. Do the applications require multi-casting or broadcasting? If so, can the communications network support these features? What effects will these features have on system performance?

Beyond message passing paradigms, the designer must also consider the physical properties of the communications path and the residual error properties of communications protocols. Sending messages between nodes will incur a delay for access to the network, for transmission of the message, and for propagation. In addition, the protocol processing software itself will add to the message delay. And these delays are generally stochastic. How can worst-case delay be computed for messages that pass between nodes? Many times messages are garbled, misordered, or lost during transit between nodes. These errors can introduce random delays when the communications protocols attempt to recover from them. What happens if the communications protocols cannot recover? Are some forms of errors acceptable in order to better bound the delay? What happens if one of the nodes fails? Can pending transactions be recovered or must they be restarted?

Another issue that sometimes occurs in a distributed system is incompatibility among data representations. To address such incompatibilities, methods exist for encoding data in a standard transfer syntax that can be recognized and decoded by all systems in the network. Of course, the processing time for

encoding and decoding the data adds to the communications delay and, thus, must be taken into account by the designer.

Another issue that appears whenever systems are distributed and accessible by a network is that of security. For a real-time system, particularly a reactive control system or an interactive system with access to confidential information, five security issues must be considered. First, a means must exist to authenticate that a message arriving from an external process does indeed originate with that external process. Second, having established the identity of an external process, a means must exist to control the access of the external process to only those resources to which that process is entitled. Third, messages exchanged between nodes on a network must be protected so that the message sent is exactly the message received, or else the receiver should be able to detect that the message has been changed. In some situations, messages exchanged between nodes might require confidentiality so that observers outside of the communicating nodes cannot eavesdrop on the conversation. Finally, in a selected set of applications, requirements might exist to prevent the sender of a message from later claiming that the message was never sent.

As the reader can readily see, when components of a design are distributed a bewildering array of issues faces the software designer. In truth, the present state of design practice is unable to cope in any general sense with distributed, real-time systems. The best that is achieved in practice today is to build a distributed, real-time system from homogeneous components, to provide dedicated communications resources between nodes, to isolate the network physically to obviate security concerns, to employ forward error correction techniques to keep communication errors within known bounds, to arrange hot standby nodes to take over when critical nodes fail, and to use simple asynchronous or RPC mechanisms to communicate between

processes on distinct nodes. Even given these restrictions, design of distributed, real-time systems remains a difficult way to make a living. This should be apparent to the reader who recalls the difficulties attendant to designing concurrent, real-time systems. Distribution, even when severely curtailed, adds to the designer's challenge.

To close this section on the design problem, the following extended quote from W. Beregi of IBM describes the state of software design practice.

We have commonly defined architecture using ambiguous natural language, diagrams, and other freeform notations. Such expression hinders our ability to communicate accurately the system's structure and prevents us from formally analyzing the structure and dynamic behavior of the system. Thus we design and implement functions based on structures and protocols that are weakly specified, poorly communicated, and not formally validated during design. We are unable to test the feasibility of our initial architecture ideas or compare alternative proposals. We are unable to examine the architecture specification and determine the effect that architecture tradeoffs and function placement decisions have on system performance, usability, and reliability. To explore these aspects, we must either create expensive, throwaway models of the system or wait until we integrate the implemented functions late in the test cycle. Costs usually dictate that few, if any, alternative designs are considered. Poor architecture decisions can propagate through all stages of a project and cause costly rework to undo design and implementation based on those decisions. [BERE84, p. 4]

Beregi goes on to observe that each new system is usually custom designed -- existing, successful designs are not reused because no ready made substructures or subassemblies exist into which new components can be fitted.

The next section examines how real-time systems are designed today. First, the question of schedulability in hard real-time systems is considered. Then two different types of

approaches to designing real-time systems are described. Within each approach, some specific design methods are surveyed.

III. Some Design Approaches

Approaches to designing real-time systems can be classified into two general categories. One category, deterministic approaches, encompasses real-time design methods that are most often used in practice and that have at least a thirty-year history. The second category, concurrent approaches, are gaining in popularity, but have only about a ten-year history of use in real-time applications. The community of researchers and practitioners of real-time design methods remains divided on which class of methods achieves the best results. Supporters of deterministic approaches argue that concurrent designs cannot ensure that application timing constraints will be met. Supporters of concurrent designs argue that deterministic approaches result in designs that are difficult to understand and maintain. Further, advocates of concurrent approaches believe that recent results in the area of rate monotonic scheduling theory can be used to ensure that concurrent designs will meet application timing constraints. These arguments are considered in more detail below.

A. The Question Of Schedulability

Most hard real-time (HRT) applications consist of periodic tasks with hard deadlines and a small number of aperiodic tasks

which require short response times. To ensure that a HRT system meets required deadlines and response times a feasible schedule must exist for the software tasks comprising the system. A feasible schedule exists if every task begins execution when enabled to run, or later, and every task still meets its established deadlines. Scheduling is complicated by the fact that certain relationships must be observed between the tasks. For example, some tasks may produce results that are needed by other tasks, thus implicitly forcing an ordering requirement among task executions. As another example, tasks that share access to resources must be kept from simultaneous access to those resources. Another consideration is that switching between tasks introduces overhead; thus, tasks should be scheduled so as to reduce preemptions. Of course, preemptive scheduling is possible only when tasks do not require mutually exclusive access to shared resources.

These HRT scheduling constraints are difficult to meet, especially in complicated systems. One approach to meeting such constraints advocates using a pre-run-time scheduling algorithm to account for all inter-task relationships and to then search for a feasible schedule that will satisfy the timing constraints of the application. [PENG93, SHEP91, XU93] First, the tasks in the system are identified and classified as periodic or asynchronous. Each periodic task is characterized with a set of parameters: period, worst-case execution time, deadline, and

release time (i.e., the delay between the beginning of a task's period and the earliest time the task can run). Each asynchronous task is characterized by a similar set of parameters: minimum time between two consecutive invocations of the task, worst-case execution time, and response time. Second, any relationships between the tasks are identified and described. These relationships typically include precedence ordering (e.g., task A must execute before task B), exclusion (e.g., execution of task C be interleaved with execution of task D), and resource constraints (e.g., task E must run on processor Y). Such inter-task relationships can become quite complex, especially in a large system of tasks running on multiple processors.

"For satisfying timing constraints in hard real-time systems, predictability of the system's behavior is the most important concern; pre-run-time scheduling is often the only practical means of providing predictability in a complex system." [XU93, p. 73] To enable pre-run-time scheduling of complex real-time systems, the task descriptions and relationships must be encoded for use by an automated search algorithm. In general, such algorithms use heuristic, branch and bound searches to seek a feasible schedule. [PENG93, SHEP91, XU93] Xu and Parnas identify and evaluate over twenty pre-run-time scheduling approaches for real-time systems. [XU93]

Advocates of pre-run-time scheduling can point to specific practices, used in concurrent designs, that reduce the predictability of a system. [XU93] One such practice is assigning static priorities to tasks (this is the only approach supported, for example, by the Ada language) and then to allocate resources in a strict priority order. Such practices can result in missed deadlines, because in certain situations a processor must be left idle, so that deadlines can be achieved, even though some task may be ready to execute. In essence, Xu and Parnas argue that pre-run-time scheduling can use global knowledge to determine a fixed schedule that will meet deadlines, while the local knowledge encoded as task priorities results in non-deterministic, run-time behavior that can cause missed deadlines.

A second practice, standard in concurrent designs, that can lead to timing problems is the use of complex run-time mechanisms for task synchronization and mutual exclusion (e.g., semaphores, locks, and monitors). Use of such mechanisms makes timing difficult to predict, incurs overhead in context switching, and can lead to deadlock and starvation. Of course, properly used in careful designs, run-time synchronization mechanisms should not cause deadlock and starvation; however, using such mechanisms can result in unpredictable waiting times.

Another bad practice that Xu and Parnas find to be common in concurrent designs is that of allowing external events to

interrupt processes and occupy system resources at random times. Such interrupts make task timing difficult to predict and incur unnecessary context switching time. Xu and Parnas argue that most internal or external events can be buffered until some periodic task can process them; thus, that a deterministic schedule can be maintained even in the face of asynchronous events.

As a final caution, Xu and Parnas assert that using stochastic simulations, as system designers often do, to verify the performance of a design is unsatisfactory. Such simulations can indicate the presence of flaws, but not their absence. Also, stochastic simulations show only average timing behavior, not the worst-case performance of the system. This view is shared by other researchers. [MAHJ84]

Advocates of concurrent designs have long held that cyclic executive approaches require application software to be divided into execution units as dictated by timing and synchronization requirements rather than by the logic of an application. As a result, advocates of concurrent designs argue that cyclic designs reduce the understandability, maintainability, and extendibility of the software. Concurrent designs, on the other hand, enable designers to manage tasking at an abstract level, divorced from the details of task execution. But, because in HRT systems these concerns are secondary, advocates of concurrent designs have been unable to convince most

practitioners that concurrent approaches to HRT systems are feasible. The recent emergence of rate monotonic scheduling theory might change this situation.

Rate monotonic theory assures

that as long as CPU utilization of all tasks lies below a certain bound and appropriate scheduling algorithms are used, all tasks will meet their deadlines without the programmer knowing exactly when any given task will be running. Even if a transient overload occurs, a fixed subset of critical tasks will still meet their deadlines as long as their CPU utilizations lied within the appropriate bounds. [SHA90, p. 53]

Rate monotonic theory consists of four theorems that specify how a concurrent system of tasks will behave. [OBEN93, SEI92, SHA90] Each theorem is considered below.

The first two theorems address scheduling for n independent, periodic tasks, each assigned a fixed priority with higher priorities going to tasks with shorter periods.

Theorem 1. n independent periodic tasks scheduled using rate monotonic analysis will always meet deadlines if:

$$\sum_{i=1}^n C_i/T_i \leq n(2^{1/n} - 1) = U(n) \text{ where}$$

- C_i is the execution time of task i ,
- T_i is the period of task i , and
- $U(n)$ is the CPU utilization of n tasks. [SHA90, p. 54]

Theorem 2. For a set of independent periodic tasks, if each task meets its first deadline when all tasks are started at once, then the deadlines will always be met for any combination of start times. [SHA90, p. 54]

Given a value for n , the bound $U(n)$ can be computed. As $n \rightarrow \infty$, $U(n)$ approaches 69%. So, for a large system the worst-case CPU utilization for rate monotonic scheduling (RMS) to hold will leave 31% of the CPU capacity unused. Deterministic scheduling with cyclic executives can achieve much higher CPU utilization and still ensure that deadlines are met. Proponents of RMS point out that 31% CPU idle time is the worst-case and that a more likely figure for a randomly chosen set of tasks is 12% CPU idle time. Further, RMS advocates argue that if $U(n)$ is exceeded, the critical time zone theorem (Theorem 2) of RMS can be used to determine if deadlines can still be met. In other words, Theorem 2 states that if any schedule can be found such that when all tasks are started together the deadlines are met, then the task set is schedulable, regardless of execution order. Rate monotonic theory expresses this as a mathematical test that is captured in a third theorem.

Theorem 3. A set of n independent periodic tasks scheduled by rate monotonic analysis (RMA) will always meet its deadlines, for all task phasings, if and only if,

$$\forall i, 1 \leq i \leq n,$$

$$\min(k, l) \in R_i \sum_{j=1}^i C_j \frac{1}{T_k} \lceil \frac{l T_k}{T_j} \rceil \leq 1 \text{ where}$$

- C_j is the execution time of task j ,
- T_j is the period of task j , and
- $R_i = \{(k, l) | 1 \leq k \leq i, l = 1, \dots, \lfloor T_i / T_k \rfloor\}$ [SHA90, p.55]
-

This theorem expresses formally the checking required by Theorem 2.

Rate monotonic theory guarantees that the n periodic tasks within the schedulable set will meet their deadlines even if the CPU is overloaded. The price of this guarantee is that some computationally expensive tasks may not fit within the schedulable set. Should a critical task not fit within the schedulable set, RMS allows such a task to be divided into a number of tasks with lower computation times and shorter periods. In this way, a critical task can be inserted into the schedulable set as a group of tasks. (Of course, artificially dividing a task into sub-units to achieve schedulability incurs the penalties of reduced understandability, maintainability, and extendibility.)

As presented so far, RMS addresses only periodic tasks; however, aperiodic tasks within a real-time system must also be scheduled to meet response time goals. Rate monotonic theory allows aperiodic tasks to be treated as periodic tasks with a period equivalent to the maximum rate at which its associated events enter the system. By modeling aperiodic tasks as periodic tasks, the rate monotonic analysis theorems can be used to schedule them.

A more difficult problem for RMS deals with task synchronization. As pointed out by advocates of deterministic scheduling, semaphores, locks, monitors, rendezvouses, and similar synchronization mechanisms can prevent a system from meeting deadlines by introducing non-deterministic delays as

tasks wait for access to resources or for a rendezvous. One way to avoid these problems is to ban preemption during critical sections. Another method, advocated by some proponents of RMS, is to implement a priority ceiling protocol. [SHA90] A priority ceiling protocol would require two conventions: 1) when a task begins to block the execution of a higher priority task, then the priority of the blocking task will be raised to that of the highest priority task that is being blocked and 2) a new critical section can start execution only if the section executes at a priority higher than the one it preempts.

If a ceiling priority protocol is implemented, then a concurrent design's schedulability can be assessed using the fourth theorem of RMS.

Theorem 4. A set of n periodic tasks using the priority ceiling protocol can be scheduled using RMA for all task phasings, if

$$(\sum_{i=1}^n C_i/T_i) + \max(B_1/T_1, \dots, B_{n-1}/T_{n-1}) \leq n(2^{1/n} - 1) \text{ where}$$

B_i is the longest duration of blocking that can be experienced by task i .

Unfortunately, most run-time systems and real-time executives do not yet support a priority ceiling protocol, although some do support an less capable priority inheritance protocol that allows a blocking task to increase its priority to the level of the highest task it is blocking. Another unfortunate fact is that many concurrent designs are targeted for implementation in Ada, yet the Ada language does not provide the support necessary

to use RMS effectively. Still, RMA can be used with Ada provided certain coding guidelines are followed and provided that a special-purpose run-time system is available that implements a priority ceiling protocol. [SHA90]

Rate monotonic analysis can be expected to have a larger role in the future because its principles have been adopted in emerging standards for FUTUREBUS+ (a hardware bus intended for distributed, real-time systems), for Posix (a standard operating system interface), and for Ada 9X (the next generation Ada language and run-time system). [OBEN93] A few vendors of Ada run-time systems and real-time executives are already offering implementations of the priority inheritance protocol. [OBEN93] In addition, work is underway to extend rate monotonic analysis to multiprocessor configurations. [JOSE86]

The reader should bear in mind the issue of schedulability as the discussion turns now to design approaches. The prime objectives for hard real-time software are: 1) a fast response to critical events, 2) a maximum number of timely transactions per second, and 3) stability under transient loads. The secondary objectives of such software include: 1) understandability, 2) maintainability, and 3) extendibility. Deterministic design approaches aim to ensure the primary objectives at the cost of the secondary objectives. Concurrent design approaches aim to maximize the secondary objectives, while still enabling the primary objectives to be satisfied.

B. Deterministic Design Approaches

In general, deterministic design approaches require that processing logic be divided into scheduling blocks that run to completion every time they are called. [FAUL88] Precedence relationships and periodicity are then defined for the scheduling blocks and a pre-run-time scheduler produces a schedule that satisfies precedence and timing constraints. The scheduling blocks are then distributed to programmers along with a maximum processing time. Each programmer must ensure that his module performs correctly and executes within the maximum time allotted. At run-time, a cyclic executive manages execution of each scheduling block in accordance with the predetermined schedule. As long as each module does not exceed its processing budget, all deadlines will be satisfied. Deadlock, starvation, and livelock cannot occur. Mutually exclusive access to shared resources is guaranteed. Of course, the software will not be very adaptable to change. As functional requirements are added, the design cycle must begin again because all of the modules in the design are tightly inter-related.

While deterministic design approaches have been used to develop real-time systems over the past three decades, no rigorous, repeatable methods have been documented. Some practicing designers employ published techniques for structured analysis and design, adapting them as necessary to meet the unique need of cyclic designs. The author can draw on his own

experiences designing air traffic control systems to illustrate deterministic design.

Design generally begins by examining periodic external stimuli to determine what information arrives at the system and how often. Then the required periodic outputs are studied to detail the content and rate of output generation. Once the periodic nature of the system is understood, asynchronous inputs are analyzed to determine how often they arrive and what processing they require. Designers who use structured analysis produce a system context diagram and a set of hierarchical data flow diagrams to document the results of the analysis.

Design continues with the layout of a common data repository that all modules can access (mutual exclusion will be guaranteed by the cyclic executive). In general, the common data repository accumulates information received from external events and includes system configuration data needed to generate output information. The general outline of the system will be: 1) process periodic external inputs and update common data, 2) generate periodic outputs, and 3) process asynchronous inputs. The system is structured logically into the modules needed for the particular application; a module ordering is established and a schedule is produced to meet the timing constraints. Finally, each module is allocated a piece of the available time. Designers who use structured design produce a data dictionary and a module hierarchy chart.

In summary, deterministic design approaches treat system design as a process of allocating available CPU time among the system modules based on the synchronicity of the input events and the update rate of the output events. Asynchronous events are budgeted to have some amount of time within the system cycle, and therefore, the system's ability to handle asynchronous events is bounded. Modules operate on common data and must live within a strict time budget.

Deterministic approaches to designing real-time systems, although practiced widely, make little use of modern software engineering techniques. Concurrent design approaches attempt to introduce modern software engineering methods into the design of real-time systems.

C. Concurrent Design Approaches

In general, concurrent design approaches involve two phases: 1) problem analysis and 2) architectural design. The objective of the problem analysis phase is to understand the structure, data, and behavior associated with an application. In general, the results of problem analysis include: 1) data and control flow diagrams, 2) state transition diagrams or tables, 3) data dictionaries, and 4) data transform logic specifications. The second phase, architectural design, uses the products of the problem analysis to create a concurrent design structured as a set of tasks and information hiding modules. In some cases, a concurrent design method also

includes procedures to estimate a system's worst-case response time to external events. In general, the results of concurrent design approaches are static structures, supported by task and module specifications, that become dynamic only after the implementation is coded.

A number of approaches to develop concurrent designs can be found in the literature. [GOMA84, HULL91, KURK93, NIEL87, NIEL90, RIDD80, SAND89a, SAND89b, SAND93, WITT85, YAMA93] The present paper discusses only a few of these.

Entity-Life Modeling (ELM) as proposed by Sanden first seeks to identify threads of events (called subjects) in the problem domain and then passive objects. The subjects are used to model resource users and the objects are used to model resources. The threads of events will become tasks in the design, while the passive objects will become information hiding modules. A major concern of ELM is ensuring mutual exclusion when tasks access resources, while also preventing deadlock when multiple tasks are competing for access to the same set of resources. In general, ELM objects are required to implement their own mutual exclusion. When a subject needs simultaneous access to a set of resources, the designer must define and enforce resource acquisition rules so that tasks do not deadlock while acquiring resources. This approach in practice means that the designer must "[e]stablish a transitive, irreflexive ordering of resources and permit the cumulative allocation of

resources only if the allocation conforms to the ordering."
[WITT85, p. 68]

ELM provides a useful model for thinking about concurrent design problems when a single processor is involved. ELM does not, however, apply when a system is distributed. This restriction arises because ELM requires an execution environment where threads of control share an address space. [SAND93]

Gomaa has proposed a family of methods for designing concurrent and distributed systems. [GOMA84, GOMA89] These methods start with a problem analysis based either on real-time structure analysis (RTSA) or concurrent, object-based requirements analysis (COBRA). When a distributed system is under consideration, Gomaa provides guidelines for logically structuring a system into subsystems that can execute on separate processors. After a subsystem is allocated to a processor, design proceeds with a problem analysis, using RTSA or COBRA, for each subsystem.

Upon completion of the problem analysis, a designer will have produced a set of data/control flow diagrams (with a state transition diagram for each control transform and a process specification for each data transform) and a data dictionary. Design continues by applying task structuring and cohesion criteria to the data/control flow diagrams to produce a task architecture diagram (TAD). Each task is also described through a task behavior specification (TBS) that records that inputs and

outputs of the task, the priority of the task, the reason that the task exists, a link to the control and data flow diagrams, and a specification of the task's control logic. Next, module structuring criteria are applied to the data/control flow diagrams to identify the information hiding modules in the design. For each module, a specification is written describing the type of module, the module operations, and the synchronization requirements for the module. The information hiding modules are then allocated to tasks and a system architecture diagram is produced. Gomaa also provides some optional steps for mapping the resulting design to the Ada language.

Nielsen and Shumate describe a concurrent design approach that aims at an Ada implementation from the beginning. [NIEL87]

Beginning with the same context diagram that usually precedes any software analysis, Nielsen and Shumate immediately assign tasks to control the devices identified on the context diagram. Next, Nielsen and Shumate decompose the middle part of the system using standard data flow diagrams. From the data flow diagrams, concurrent processes are identified using a set of heuristics. Then interprocess communications mechanisms are defined, followed by any intermediary Ada tasks needed to implement decoupled inter-process messages. (Ada tasks can communicate only via rendezvous and thus intermediary tasks are

needed when two applications task must communicate via loosely-coupled messages.)

Once tasks have been identified, Nielsen and Shumate move immediately to package the tasks in Ada and then to specify those packages using Ada as a program design language. After the Ada package specifications are written, the Nielsen and Shumate design methods requires two design reviews, followed by an update to the design documents. Although the Nielsen and Shumate method is not intended for distributed system design, Nielsen later explored some of the issues involved in designing distributed systems. [NIEL90]

While concurrent design approaches generally lead to designs that are easy to understand, maintain, and extend, some shortcomings can be identified. For one, typical design approaches lack semantic meaning prior to reaching the code level. [KURK93] Also, most concurrent design approaches lack support for timing analysis, particularly where task synchronization is involved. These deficiencies leave a designer unable to assess the dynamic behavior of proposed designs.

IV. Open Issues In Designing Distributed, Real-Time Systems

The preceding sections of this paper considered the challenges facing designers of distributed, real-time systems and surveyed some available approaches for designing such

systems. Comparing the challenges with the available approaches reveals that some issues remain unresolved. In this section, the most critical open issues are identified and briefly explained.

One category of open issues results from the nature of software requirements documents. The requirements for most software systems, including real-time systems, are expressed in natural language. Indeed, for large systems, requirements specifications are usually written by a group of individuals, each writing in their own style about a particular aspect of the software requirements. The use of natural language by multiple authors results typically in a requirements specification that contains inconsistencies, ambiguities, and omissions. For those researchers addressing requirements engineering topics, the open challenge is to find effective methods to reduce, and eliminate if possible, these defects from the software requirements specification. *For researchers addressing software design, however, the open challenge is to find methods to detect and resolve flaws contained in software requirements specifications.* Software designed to meet flawed requirements will not satisfy the customer, and the designer will be held responsible for these failings.

A second open issue involving requirements is particularly germane to real-time systems. *Available methods for specifying software system timing requirements are inadequate.* In general, system timing objectives are treated as nonfunctional requirements, expressed in a probabilistic fashion using natural language. Such treatment appears inappropriate for hard real-time systems because certain timing objectives represent deadlines that bound functionally correct behavior. A hard real-time system that performs all functions correctly can still fail if a single deadline is missed. Improved methods must be found for describing deadlines and response time requirements

for hard real-time systems. Should deadlines and response times be related to devices? Should response times be related to scenarios of events that map an external input to an external output? How should the system load be characterized? Should the system load be expressed as the set of individual loads generated by external inputs? Should timing requirements be expressed as maximum response times given a worst-case system load? These are only some of the issues on which no agreement exists.

A second category of open issues for designers of real-time systems arises when the software is distributed throughout a network of nodes. From among all of the special considerations for distributed systems, as discussed in section II of this paper, two appear unavoidable, yet difficult to resolve. First, the properties of the communication paths and protocols in a distributed system introduce stochastic delays and residual error probabilities into inter-task communications. *Methods must be found to bound the maximum communications delay and residual error rate between nodes in a distributed, real-time system.* Without such methods, a designer cannot possibly ensure that task deadlines and response times will be satisfied. The best that can be achieved with current methods is some assurance that, within the bounds of a known load, communication delays and residual errors will not exceed a specified value with some probability. Perhaps the only realistic means to address this challenge will involve raising an exception when a required delay or error rate is exceeded. This introduces the second unavoidable challenge faced by designers of distributed systems: choice of inter-task communication paradigm.

Designers of real-time systems are usually forced to adopt the inter-task communications conventions available with the real-time executive or the language run-time system used for the implementation. In general, the available primitives will also

integrate external device interrupts into the conventions for inter-task message exchange. When a system must, however, be distributed among multiple nodes, few, if any, real-time executives or language run-time systems provide native mechanisms to handle inter-node message exchange. The designer then must define mechanisms for inter-node message exchange and must establish the relationships between these mechanisms and the mechanisms for local message exchange. Further, the designer must include these non-application functions within the system design and then ensure that they are properly implemented for each type of node in the distributed system. *The open challenge for researchers is to remove this burden from designers by developing an effective paradigm for distributed, real-time, inter-task communications.*

The third category of open issues for designers of real-time systems stems from the static nature of design documents. Most design methods result in diagrams and supporting paper specifications that clearly express the structure of a design. Some automated tools even allow consistency checking among the various interrelated pieces of a design. Unfortunately, because most design methods lack a formal semantics, dynamic evaluation of designs is usually postponed until system testing. Redesigning after flaws are found during system tests usually comes with a high price tag.

The challenge, then, for researchers is to devise methods to enable designs to be verified dynamically before a system is implemented. Such methods should enable designs to be checked for safety properties -- absence of deadlock, livelock, unfairness, and failure, as well as presence of mutual exclusion, proper synchronization, boundedness, and resource conservation. In addition, verification methods should enable designers to assess resource utilization and to predict the performance properties of the design. A means should be

included to map the design onto various hardware and network configurations and to assess the effects of these mappings on system performance and correctness. To be most effective, dynamic verification of designs should proceed directly from the design documentation associated with a design method.

The reader can probably identify other open issues arising from the material presented in sections II and III, but the author believes that the most critical challenges are those outlined above: improving the designer's ability to specify and analyze requirements, devising a method to bound the uncertainties and to mask the complexities associated with inter-node communications, and enabling designs to be verified dynamically before they are implemented. To address these challenges, researchers are investigating a number of formal methods and models, as well as related languages. Some of the more prominent formal methods are considered in section V. Section VI surveys a number of design languages based on formal models. Section VII examines several attempts at constructing design environments that integrate complementary tools in an effort to meet the challenges facing software designers.

V. Formal Methods For Designers

Formal methods appear to promise effective solutions to the open issues that challenge designers of distributed, real-time systems. Formal methods encompass models, typically supported by a notation, that rest on a sound mathematical basis. [WING90]

By encoding critical aspects of a system into a formal model or description, designers can uncover ambiguity, incompleteness, and inconsistency in requirements. Formal methods, when supported by appropriate tools, can also be used to verify system designs.

Two general categories of formal methods can be defined: 1) behavioral methods and 2) structural methods. [WING90] Behavioral methods allow a designer to describe formally the intended behavior of a system and then to investigate various properties that the system will exhibit during operation. For designers of concurrent systems, behavioral methods address issues of task sequencing, synchronization, mutual exclusion, and, with some methods, task timing and performance. Some examples of behavioral methods (covered below) include finite state automata, Petri nets, temporal ordering, and modeling and simulation.

Structural methods enable designers to express formally the properties that a correctly behaving system will exhibit and, in some cases, to provide proofs that an underlying implementation will exhibit the expressed properties. Structural methods allow designers to specify invariants for information hiding modules, as well as preconditions and post-conditions for each module operation. With some methods, a designer can even specify the invariants and pre and post-conditions for procedural code. Some examples of structural methods (covered below) include temporal logic, axiomatic methods, and abstract data types. In general, structural methods have proven labor intensive, have yielded inefficient proofs, and have been difficult for the average software designer to master. [HOAR87]

The paragraphs that follow examine, one by one, some promising formal methods. Behavioral methods are considered first, followed by structural methods.

A. Finite State Automata

Finite State Automata (FSA), also called Finite State Machines (FSMs), represent system behavior as a set of states. A FSA can be in only one state at a given moment, but can change states in response to external events. Such a model enables a system to order its behavior in the face of random asynchronous

events that may arrive from many sources. Pure FSA, in most practical applications, require a large number of states to

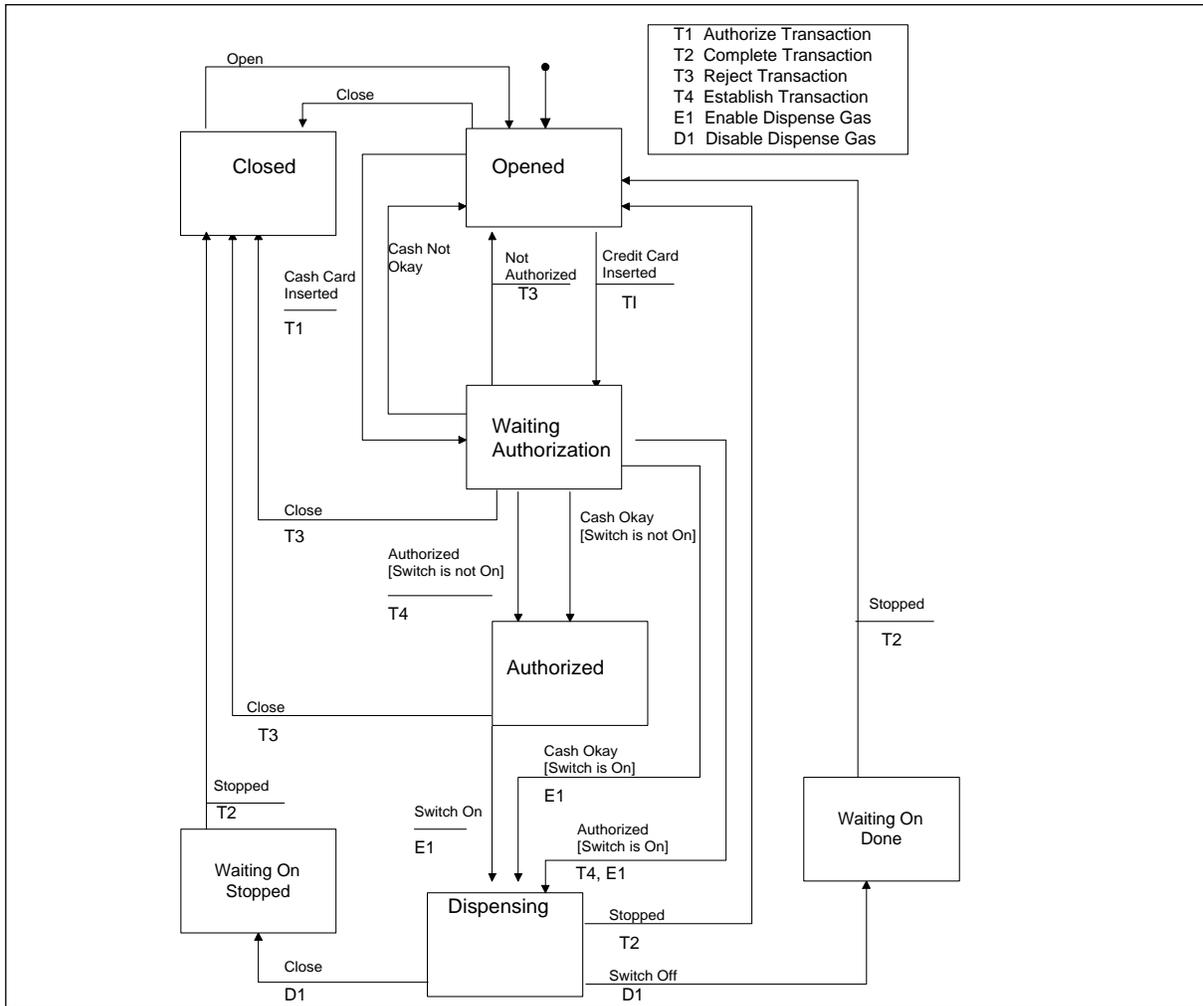


Figure V-1. Example Finite State Automata

properly describe a system's response to external events. To reduce this state-explosion problem, must useful FSA models have been augmented to include predicates that guard the activation of transitions between states based on historical information that is retained in state history variables. Perhaps these points will become clear through considering an example. An example can also illustrate the graphical notation, called state transition diagrams, typically used to represent FSA.

Figure V-1 shows a state transition diagram representing the FSA for an automated gas pump. Each rectangle enclosing a label represents the state named by the label. The set of states in the example is: (**Opened**, **Waiting Authorization**, **Authorized**, **Dispensing**, **Waiting On Done**, **Waiting On Stopped**, **Closed**) Transitions between states are shown as directed arcs with the arrow head pointing to the new state and away from the old state. A single arc without an old state identifies the initial state of the FSA (**Opened** in the example). Each transition is triggered by the arrival of an event. The set of events in the example is: (Open, Close, Cash Not Okay, Not Authorized, Credit Card Inserted, Cash Card Inserted, Cash Okay, Authorized, Switch On, Switch Off, Stopped). Each transition can have an associated set of actions that are considered to occur instantaneously as the transition fires. In the example, the set of actions is given in a box within the figure. Each action has a short label (e.g., T1, D1) and a descriptive name (e.g., Authorize Transaction, Disable Gas Dispenser). Also shown are examples of predicates that guard a transition. In Figure V-1, predicates are bounded within square brackets (e.g., [Switch is On]). Consider the state **Waiting Authorization**. When the Cash Okay event arrives, two transitions are potentially enabled: one transition moves the FSA to the state **Authorized** and is guarded by the predicate [Switch is not On] and the other transition moves the FSA to the state **Dispensing** and is guarded by the predicate [Switch is On].

FSA, described with state transition diagrams, are typically used to prescribe an acceptable sequential order in which arriving external events can be processed. While state transition diagrams are most convenient for human comprehension, other forms of FSA representation are better suited to machine processing. Commonly used encodings include: nested, state-X-event, case statements in high-level programming

languages and state-X-event tables that drive an FSM interpreter. [KUUL91] Some high-level languages have been modified to include constructs that directly represent FSMs. [ISO92]

Once an FSA is described in a machine-processable form, event-state-transition tracing tools can be used to verify that the FSA captures the desired behavior; however, when extensive use is made of predicates, the FSA must be translated into an FSA that is predicate-free prior to applying the tracing tools. The resulting FSA may explode into hundreds of thousands of states and, thus, prove computationally difficult to verify. For example, extended finite state machines are sometimes used to specify the allowable behaviors in telecommunications systems, and then automated tools are applied to generate scenarios for system tests. [CAN85] In these cases, human intervention is required to eliminate the generation of duplicate tests and to prune the resulting test set to an acceptable size.

FSA enable concise specification of allowable sequences of behavior in a form that is comprehensible to humans, yet that can be translated straightforwardly into a machine-processable encoding. All of the described behavior is sequential and applies to a flat, single task. No provision exists for describing timing nor concurrency among events. These shortcomings were addressed by researchers through a set of extensions to FSMs that were proposed gradually over two or three decades.

A first extension involved using multiple, communicating finite state machines to model interactions between cooperating sequential tasks. This enabled concurrent tasks and task synchronization to be modeled simply and efficiently. Inter-task communications were then represented as asynchronous events exchanged between FSMs. Events arriving at a FSM were

simply deposited into a FIFO queue and then processed one-at-a-time. A second extension allowed each state in a FSA to be modeled with a nested FSA. Introducing hierarchies of FSMs assisted designers in managing complexity by allowing large systems to be represented as compositions of simple FSMs, rather than as a single, huge FSM. Combining the first two extensions permitted intrastate concurrency to be modeled by allowing multiple FSMs to execute under that control of a parent FSM that was itself embedded in the state of its own parent. As the number of cooperating FSMs to be modeled increased, the difficulties of communication and synchronization between them increased as well.

To harness the many extensions to FSMs and to introduce some discipline into inter-FSM communications, a number of formal models were developed during the 1980's. One such model, Extended State Transition Language (Estelle) became an international standard for describing communication protocols and distributed systems. [ISO92] This model will be considered in some detail in section VI when design languages are considered. For now, Estelle can be understood as a model based on communicating, finite state machines which exchange events asynchronously through unidirectional channels. The communicating FSMs can operate as peers or in a parent-child relationship. Using the Estelle model, a number of inter-task arrangements can be represented and then exercised through a run-time environment.

Another model, Communicating Real-Time State Machines (CRSM), provides a complete, executable notation for specifying real-time systems. [SHAW92] Event exchange between state machines is modeled as synchronous communication across unidirectional channels (along the lines of CSP, a language described in section VI). CRSM also includes a novel set of facilities for describing timing properties. Each transition

action has an execution or synchronization time associated with it. Each FSM is augmented by a real-time clock machine that has access to a global time source. An underlying operational semantics for executing a CRSM system manages the firing of transitions and the modeling of the time intervals. CSRM does not permit shared data. Although CSRM is one of the few FSA models to include time, there are no facilities for structuring a system of FSMs into higher level entities (a strength, for example of Estelle) or for modeling interrupts (a common shortcoming with FSMs because each transition is atomic).

Another advanced model based on FSA, called statecharts, was invented by Harel in the 1980's. [COLE92 ,HARE87, HARE90] Statecharts define a formal semantics for an advanced version of extended FSA. The advanced capabilities include hierarchical nesting of FSA within individual states, concurrent execution of multiple state machines within a single state, and broadcast communication of events so that any event output from an FSM in a statechart will be immediately visible to every other FSM in the same statechart (and external events arriving into a statechart are also visible to all FSMs in the statechart). Statecharts also allow for non-determinism and timed transitions. Statecharts appear overly rich in features and semantics, making them difficult for a designer to use and to understand. To overcome some of these difficulties, Harel has proposed a set of tools called Statemate.

Statemate enables a designer to graphically specify, analyze and design large, complex, reactive systems. [HARE90] Although the notation is graphical, the syntax and semantics are formal. A Statemate system description comprises three views: structure, function, and behavior. Each view can be expressed with a separate graphical language. The behavior language is statecharts. [HARE87] The system structure is described as a hierarchical decomposition of modules and the information flows

between them. The structure language is called modulecharts. The functional view is drawn, in a language called activitycharts, as a set of data flow diagrams. From the combined descriptions of a system, Statemate can simulate the system's behavior or generate code to implement the system. The simulator can be used to evaluate reachability, to identify non-determinism, to detect deadlocks, and to profile transition usage. The testing performed is truly a simulation and so a designer can find errors but cannot prove the absence of errors. Using Statemate for exhaustive testing is not feasible for most real designs.

Statemate appears to provide a simulation capability to support a real-time structured analysis view of system design, but with a more powerful representation of control transforms. Some recent research aims to couple statecharts with object-oriented concepts in order to marry the behavior modeling capabilities of statecharts with the information modeling concepts of object-oriented design. The result is called Objectcharts. [COLE92]

Objectcharts are an extended form of statechart that characterize the behavior of a class as a finite state machine. The design model in which objectcharts are embedded consists of a configuration diagram (describing every object in a system by its required and provided services) and an objectchart that is similar to the object notation used by Rumbaugh, Booch, or Coad. The innovation of objectcharts is to use statecharts to represent object services that change the state of an object. Services that do not change an object's state are not described with statecharts. System behaviors can be generated by combining individual object behaviors. Objects communicate using infinite, FIFO queues to hold incoming events. The behavior of each object can be specified using a trace of incoming events and resulting output events. Included in the

system model is an intuitive definition of subtyping for objectcharts, i.e., descendant classes may be specialized by: 1) adding a state/transition that corresponds to a new service, 2) strengthening the guard for a transition, and 3) strengthening the invariant for an object.

From this description of the nature of FSA and some recent research advances and supporting tools, the reader should come away with a number of impressions. First, FSA can be used to represent cooperating tasks by describing each task with one FSA. Second, FSA do not generally include the notion of time. Third, extended features such as guarded transitions and hierarchical nesting of FSA enable a designer to better deal with complexity; however, when such extended FSA are expanded and flattened to facilitate machine analysis, state explosion can make computational verification infeasible. Fourth, no agreement exists among researchers as to how inter-FSA communication should be modeled. Fifth, inter-FSA concurrency can be modeled, but the synchronization between concurrent FSA depends to a large extent on the specific model used for inter-FSA communication. Sixth, researchers are just beginning to investigate means for integrating FSA into object-oriented design models.

Another formal tool for modeling behavior includes FSA as a subset. This tool, called Petri nets after its inventor, Carl Petri, is discussed next.

B. Petri nets

Petri nets can be used to model finite state machines, as well as concurrent activities, dataflow computation, communications protocols, synchronization control, and, if inhibitor arcs are allowed in the Petri net (PN), producer-consumer systems with priority. [MURA89] A major strength of PNs is their support, when computer-assisted tools are used, for

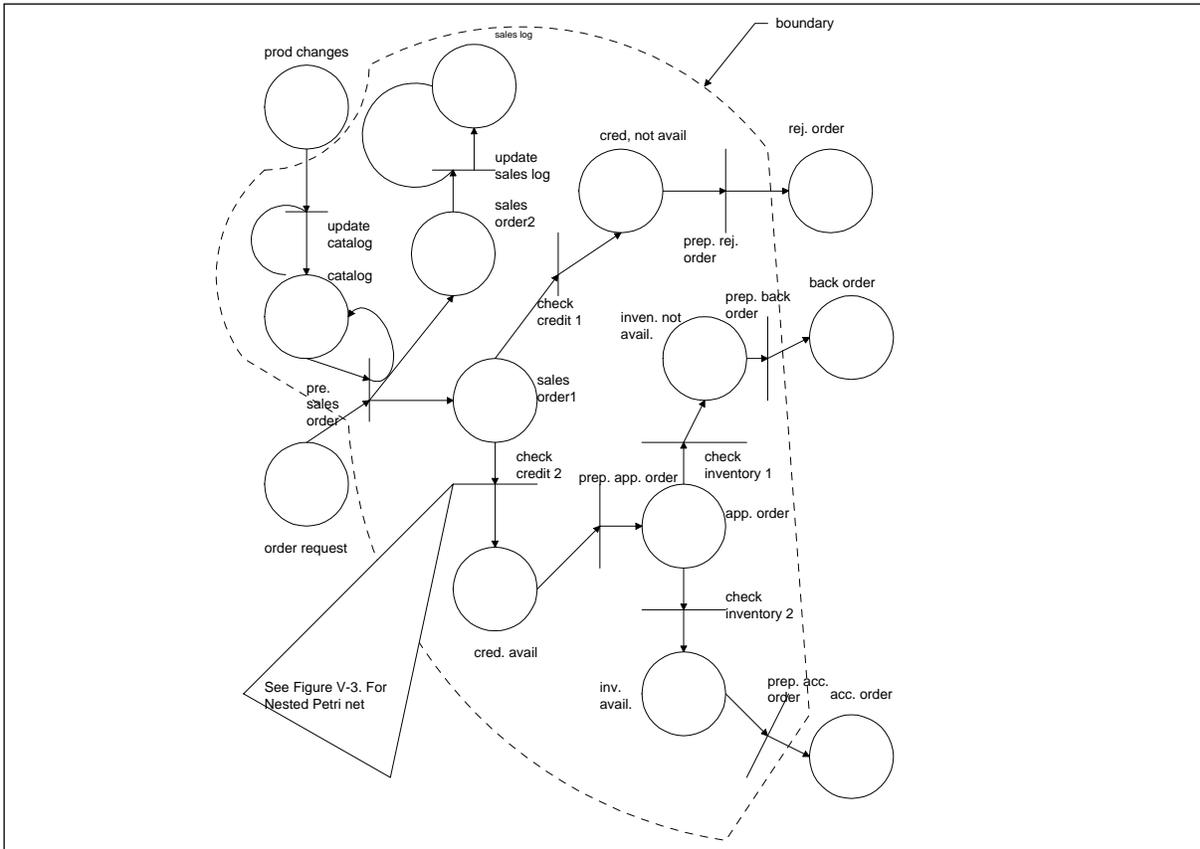


Figure V-2. Petri net Model Of An Order Processing System
[SAKT92, p.226]

analysis of many properties and problems associated with concurrent systems.

PNs can be viewed as a 6-tuple, such that $N = (P, T, E, M_0, K, W)$, where P is the set of places, T the set of transitions, E the set of arcs, M_0 the initial token marking, K the capacity function, and W the weighting function. [MURA84] P and T are disjoint. $M_0(p)$ yields the number of tokens initially at place p . $K(p)$ yields the token capacity of place p . $W(e)$ yields the number of tokens transmitted along arc e . In a so-called ordinary PN all arcs have a weight of one. In an ordinary PN, a transition is eligible to fire when a token is present at each of the transition's input places. Firing a transition results in moving a token from each of the fired transition's input places

and placing a token in each output place associated with the transition. Whenever multiple transitions are eligible to fire, one is selected non-deterministically. (Note that firing rules are different for various forms of PNs.)

PNs can be represented in a graphical form that enables a human being to visualize the behavior represented by the net. A sizable example of a graphic PN is given in Figure V-2 where the control behavior of a generic order processing system is illustrated. In Figure V-2, the dashed line represents the boundary between the order processing system and its environment. The system has

two external inputs, the places **prod. changes** and **order request**, and three external outputs, the places **rej. order**, **back order**, and **acc. order**. Each place in the order processing PN of Figure V-2 represents some system data and each transition represents some system function. The

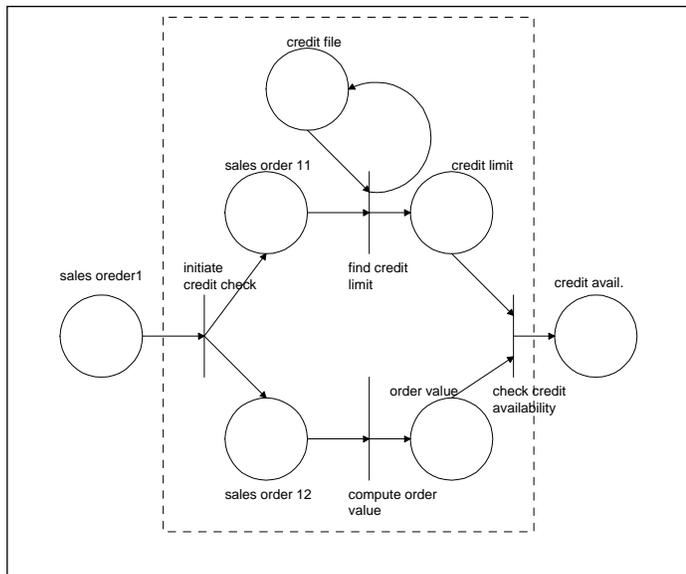


Figure V-3. Nested Petri net for Check Credit 2 [SAKT92, p. 226]

initial marking of the PN probably includes tokens in the places **catalog** and **sales log**, as these are permanent data repositories associated with the order processing system.

When a token arrives at **prod. changes** the **update catalog** transition fires, returning a token to the **catalog**. Note that a token could also arrive at **order request**, enabling the transition **pre. sales order**. Should a token arrive simultaneously at both **prod changes** and **order request**, one of the

two transitions would be selected to fire and then the other.

This arrangement of transitions represents mutual exclusion between the functions *update catalog* and *pre. sales order*. When *pre. sales order* fires, a token is removed from the places *catalog* and *order request* and tokens are entered at the places *sales order2*, *sales order1*, and *catalog*.

The existence of both places *sales order2* and *sales order1* represent the fact that when an order request is received two actions can take place concurrently: the sales log can be

Table V-1. Classifications of Ordinary Petri nets

Ordinary PN	All arcs are of weight one.
State Machine	Ordinary PN where each transition has exactly one input place and one output place.
Marked Graph	Ordinary PN where each place has exactly one input transition and one output transition.
Free-Choice Net	Ordinary PN where every arc from a place is either a unique outgoing arc or a unique incoming arc to a transition.
Extended Free-Choice Net	When two sets of places P1 and P2 intersect, the implication is that P1 = P2.
Asymmetric-Choice Net	When two sets of places P1 and P2 intersect, the implication is that P1 is a proper subset of P2 or P2 is a proper subset of P1.

updated and the order can be processed. (In PN's, two arcs leaving a transition denote parallelism.)

Consider now what occurs in Figure V-2 when a token arrives at *sales order1*. Here, two arcs leave the place. In PN's, this represents a decision because only one of the two transitions

Table V-2. Petri net Analysis Properties

Reachability	Can all token markings be reached?
Boundedness	Are the number of tokens in each place finite?
Liveness	Will at least one transition always be enabled?
Reversibility	For every possible marking, can the initial marking be reached?
Coverability	Can all potential markings be reached?
Persistence	For every transition, does the firing of the transition disable another transition?
Synchronic Distance	A measure of how often the firing of one transition is related to the firing of another.
Fairness	A measure of how often transitions get to fire relative to other transitions.
Structural Liveness	Does there exist a live initial marking?
Control-lability	Can any marking be reached from any other marking?
Structural Boundedness	Is the net bounded for any finite initial marking?
Conservative-ness	Are all initial tokens conserved?
Repetitiveness	Is there an initial marking such that every (or some specific) transition occurs infinitely often in a firing sequence?
Consistency	Is there a firing sequence such that every (or some specific) transition occurs at least once?

can fire, and when one does the other will be disabled. The two arcs leaving **sales_order1** represent the cases where: 1) a customer has insufficient credit and thus the order is rejected

and 2) a customer has sufficient credit and thus the order can be further processed.

Another feature of PNs is illustrated in Figure V-2 at the transition **check credit2**. Here a transition is represented by another PN in a hierarchical fashion. The PN that substitutes **it2** is shown, bounded by a dashed rectangle, in Figure V-3. Here a **credit file** possesses a permanent token. Upon entering the transition, two parallel paths are taken: one finds the customer's **credit limit** and the other determines the **order value**. Before leaving the nested transition, the **credit limit** and **order** for **check cred value** are synchronized as inputs to the transition **check credit availability**.

Returning to the large PN of Figure V-2, the reader should note that, though the two paths out of **sales order1** are mutually exclusive, no information exists at place **sales order1** to enable the appropriate path to be determined. This should remind the reader that PNs allow the representation of valid behaviors, but do not specify what conditions will cause which of the valid behaviors to occur.

Ordinary Petri nets can be restricted to limit the range of systems that can be modeled. A summary of the classes of restricted PNs as related to ordinary PNs is shown in Table V-1. State machines allow no concurrency nor synchronization to be represented. Marked graphs cannot depict choice because they allow no conflicts between enabled transitions. Free-choice nets cannot show confusion (i.e., a mix of conflict and concurrency). Asymmetric choice nets allow an asymmetric confusion, but disallow symmetric confusion.

PNs can be subjected to a number of analyses as indicated in Table V-2. [MURA89] To investigate the properties listed in Table V-2, three analysis methods are generally used. One common method involves generating a coverability tree. When a PN is unbounded, its associated coverability tree will become

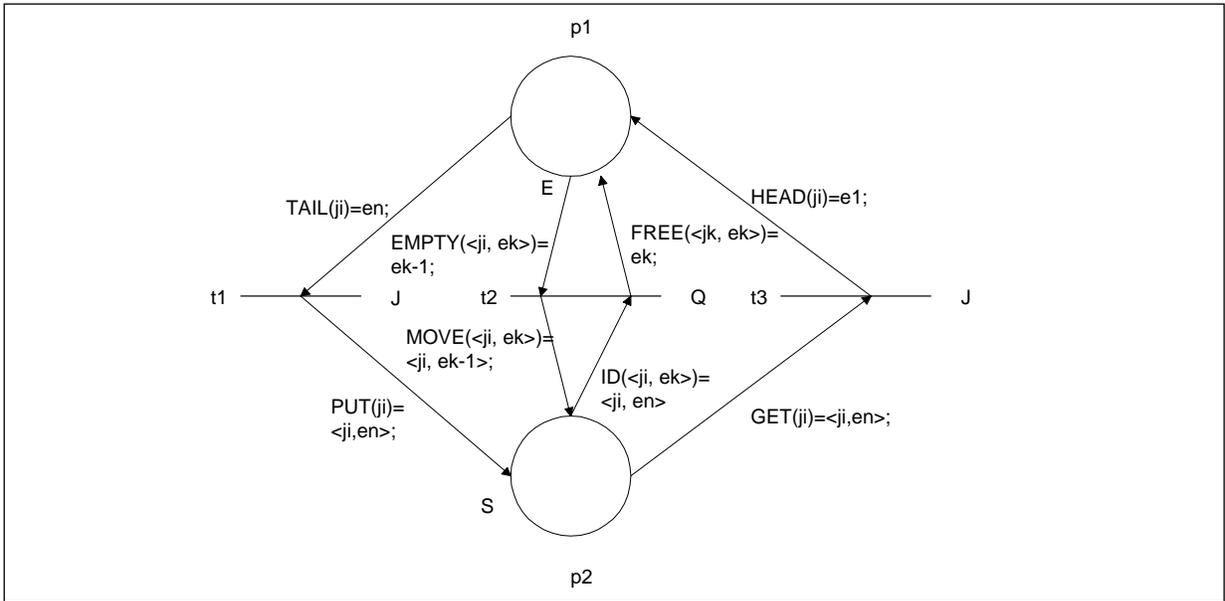


Figure V-4. High Level Petri net Depicting A FCFS Job Queue [DOTA91, p. 500]

infinitely large. To prevent this infinite path explosion, most analysis methods introduce a symbol to represent infinite behavior and then curtail each branch of the coverability tree once an infinite behavior is recognized. A second analysis technique represents PNs as an incidence matrix, coupled with a set of state equations. This technique is somewhat limited due to the non-deterministic nature of PNs. A third analysis method represents a PN as a set of simple reduction rules, a less complex abstraction that still captures the behavior encoded in the PN. All of these analysis methods are limited by the natural complexity inherent in PNs. A "...major weakness of Petri nets is the complexity problem, i.e., Petri-net-based models then to become too large for analysis even for a modest-size system." [MURA89, p. 542.] In addition, graphical PN models usually prove inconvenient when used to specify the behavior of large systems. To overcome this inconvenience, a class of nets called High Level Petri nets have been proposed by several researchers.

High Level Petri nets (HLPNs) raise the abstraction power of PN's by attaching semantic distinctions to tokens (for example, giving them types, sometimes called colors) and by associating predicates with incoming and outgoing transition arcs that allow typed tokens to be manipulated when a transition fires. HLPNs may be called predicate/transition nets or colored PN's depending upon the exact nature of the extensions. An example of a colored PN, shown in Figure V-4, can illustrate some of the concepts involved.

In Figure V-4, places represent free slots (p1) or full slots (p2) in a job queue. Transitions depict adding a job to the queue (t1), advancing a job one place ahead in the queue (t2), or removing a job from the queue (t3). Tokens come in four colors: $E = \{e_k \mid k = 1, 2, \dots, n\}$, where e_k indicates that the k th place in the queue is empty; $J = \{j_i \mid i = 1, 2, \dots, p\}$, the set of jobs; $Q = \{\langle j_i, e_k \rangle \mid i = 1, 2, \dots, p; k = 2, \dots, n\}$, where a token of $\langle j_i, e_k \rangle$ indicates that job i occupies slot k in the queue; $S = \{\langle j_i, e_1 \rangle \mid i = 1, 2, \dots, p\}$, where job i occupies the first slot in the queue. The initial marking of the HLPN finds no tokens in p2 and n tokens (one for each color $e_1 \dots e_n$) in p1.

Transition t1 is enabled if p1 contains a token that satisfies the predicate $TAIL(j_i) = e_n$ (i.e., the last slot in the queue is empty). When t1 fires, a token of color e_n is removed from p1 and a token of color $\langle j_i, e_n \rangle$ ($PUT(j_i) = \langle j_i, e_n \rangle$) is put at p2. Transition t2 depicts the movement of jobs in the queue. When a slot becomes empty, t2 fires, freeing slot e_k and moving job j_i to slot e_{k-1} . Transition t3 fires to remove a job from the queue.

When the number of colors is finite, a HLPN can be considered to consist of a structurally folded version of a regular PN. This ability to transform HLPNs into ordinary form is crucial to the analysis of the net because HLPNs cannot be

subjected to the same analytical methods as ordinary PNs. [PAPE92] The reader should notice, from reviewing Figure V-4, that, while possessing a useful level of specification convenience, HLPNs sacrifice some of the visual power of ordinary PNs. [PAPE92] Another shortcoming of HLPNs, a shortcoming shared with ordinary PNs, is an inability to model time. Several researchers investigate methods for representing time in PNs.

Two basic approaches exist for including time into PN models: 1) timed PNs and 2) time PNs. [BERT91] Timed PNs associate a firing duration with each transition. Time PNs allow two numbers, (a, b) , to be associated with each transition, where a , ($a \geq 0$), is the minimum time that must elapse from when a transition is enabled until it fires and b , ($0 \leq b \leq \text{infinity}$), denotes the maximum time during which a transition can be enabled without being fired. Time PNs are more general than timed PNs and, thus, most researchers work with time PNs. [BALB92, BERT91, GHEZ91, YAO89]¹ In general, time is used in PNs in two ways. One way depicts time delays as deterministic values and the other way, so-called stochastic PNs (SPNs), depicts time delays as probabilistic values. SPNs can be mapped into Markov chains. [MURA89] SPNs have been extended to a class of generalized SPNs (GSPNs) to manage the state space explosion that occurs with complex PNs. [BALB92, MURA89] The specific approach used will depend upon the performance properties of interest to the analyst.

¹ Note that other lesser used approaches to adding time to PNs also exist. [GHEZ91] For example, an idle time value can be attached to each place, requiring that input tokens become available only after the idle time has passed. As another example, clock mechanisms can be modeled by means of predicate/transition PNs. These are not described in the present paper, and the reader should rest assured that not every extant variation of PNs, timed or otherwise, is covered here.

Deterministic timed PNs are used to estimate the minimum time needed for each major path or cycle through a system. [YAO89] To apply this approach, the analyst must first develop a PN model from a detailed statement of the system logic. Then the PN must be reduced to only the places and transitions affecting the performance of the system. The reduced PN must then be organized into the major control cycles of the system. Once specification, reduction, and organization are complete, time PNs can yield estimates for the best- and worst-case cycle time for the system. When applying these techniques, Yao found that automatic and semiautomatic tools must be developed to aid in the analysis of large systems because PNs do not scale up well. [YA089]

Berthomieu and Diaz have proposed using enumerative analysis to simultaneously model behavior and analyze properties of time PNs. [BERT91] Unfortunately, they have encountered some of the same limitations cited by Yao. "As enumerative approaches for analyzing Petri nets can produce large sets of classes, even when the net is bounded, Petri net experts must be able to create nets with manageable numbers of classes when this can be done." [BERT91, p. 271] Berthomieu and Diaz also point out that reachability and boundedness determination for time PNs are undecidable. One other limitation of their approach is that firing transitions takes no time to complete, and so if the firing time is important, then it must be included in the time label for at least one timed transition. This adjustment tends to obscure the natural representation of the system timing model.

While deterministic time PNs have been used with some success, stochastic PNs are more controversial. [MURA89] Acceptance of GSPNs is limited by the difficulty in constructing appropriate models and by the complexity inherent in the solution of the models. [CHIO93] Balbo, et al., illustrate how

colored, generalized SPNs (CGSPNs) can be used to study both the correctness and performance of a system. [BALB92] They apply CGSPNs to study Lamport's concurrent algorithm for providing mutual exclusion on CPUs that lack an atomic test-and-set instruction. They discovered several areas that need further investigation. For example, a PN model may be too complex to analyze exactly for large numbers of basic colors. Further, the labor involved in mapping a problem to a colored PN and then to a GSPN is too great and too error prone; thus, they advocate investigation of tools for automatic generation of a CPN from a system specification and also further research regarding formalization of techniques for deriving correctness proofs.

Chiola, et al., also investigate how PNs might be used to analyze both behavior and performance, but their approach begins with a GSPN and then attempts to derive some behavioral PNs by eliminating timing from the GSPN model. [CHIO93] The novelty of their approach is based on their claim to have obtained the first structural analysis of PNs with both priorities and inhibitor arcs. They also identify some problems inherent in PNs, generally, and GSPNs specifically. For one, real system models result in graphically complex PNs. For another, the complexity of GSPN models of real systems appears too great to allow feasible modeling at a reasonable cost.

Perhaps the most ambitious attempt to integrate the behavioral, functional, and time representation in PNs is reported by Ghezzi, et al. [GHEZ91] Ghezzi introduces environment/relationship nets (ERnets), HLPNs that can be used to specify control, function, and time. The main idea behind ERnets defines tokens as environments that are really functions associating values to variables. An action is associated with each transition. Each action can also describe the input environments required by the transition and the output environments produced. Time is introduced to ERnets by

including a timestamp with each environment. Whenever an action is invoked (by firing a transition), the action must produce a value for the timestamp associated with the output environment. Three simple rules must be observed for producing the new timestamps: 1) the output timestamp must exceed the timestamps of the input environment, 2) each output environment produced by the same transition firing will have the same timestamp, and 3) transition firings in sequences must produce monotonically non-decreasing timestamps.

From the basic time rules introduced into ERnets a number of PNs with time added can be modeled. The most basic net, called Timed ERnets (TERnets), enforces a weak time model where enabled transitions are not required to fire. A variant, called Strong TERnets (STERnets), requires that an enabled transition must fire within its due time (i.e., no other firing can disable a transition). Another variant, called TBnets, represents a particular case of TERnets, but where only tokens (not environments) are timed. TBnets are introduced so that Ghezzi can show how existing approaches to adding time to PNs can be modeled with TBnets (and thus that his work represents a superset of all other approaches).

While Ghezzi's approach provides some remarkable integration and extension of existing PN models, the reader should already suspect that some drawbacks exist. For one, the main analysis aid is a tool for executing ERnet specifications. With such a tool, problems can be detected, but the absence of problems cannot be proven. Ghezzi shows that proving the properties of ERnets is undecidable. He does point out that, by ignoring tokens, standard PN analysis techniques can be applied to give an approximate analysis.

The reader will have noted that Ghezzi, and many of the researchers and practitioners discussed above, cite the need for automated assistance. For complex PNs, exhaustive analysis, in

order to establish the existence of specific properties, is often deemed computationally infeasible. For that reason, much of the practical research regarding PN tools aims at methods for creating PN models and then for exercising those models to detect errors or to characterize performance. In the following paragraphs, some of this research is reviewed.

One interesting tool, the Expert System-based PN Simulator (ESPNET), takes a timed PN as input and produces a simulator (in the OPS5 language) as output. [DUGG88] The PN can then be exercised. ESPNET, positioned as a pre-simulation tool, has been applied to generate rapid prototypes of flexible manufacturing systems.

Another approach allows a user to specify a system of hierarchical, colored PNs, to indicate the commands to and events from outside the system, and to describe guards on arcs flowing into transitions. [MICO90] Then, using a TOol for RAPid prototyping (TORA), the user can exercise his system specification. TORA consists of three subsystems: 1) an interpreter of colored PNs, 2) a hierarchical user interface, and 3) a flexible manufacturing system environment to simulate the system's external world. TORA permits parallelism to be represented visually, and enables a specification to be manipulated symbolically. All communication between PNs, and between PNs and the environment, are modeled asynchronously. TORA is another tool that enables a PN to be exercised, rather than analyzed.

Another approach applies hierarchical PNs to animate data flow diagrams (DFD). [LAUS89] Here, each leaf node of a given DFD is treated as a PN. Then, as required, terminators in one PN are mapped to initiators of another PN. The outside world is also mapped to appropriate initiators and the DFD's terminations are mapped to the outside world. Using this approach, PNs enable a DFD to be exercised.

PNs are often implemented using some form of Prolog. Recent research shows that flat concurrent Prolog (FCP) can be used to execute directly HLPNs. [DOTA91] FCP can also be used to simulate hierarchical systems of PNs and to describe and simulate timed PNs. These models can also be integrated. Since all PNs that are represented with FCP are guaranteed to be finite, some analysis can be performed on such PNs. Achievable analyses include: 1) loop-free-ness, 2) decision-free-ness, 3) consistency, and 4) determining whether a PN is synchronous or asynchronous. In addition, dynamic execution of an FCP PN can assess: 1) reachability, 2) ambiguity, 3) boundedness, 4) resource conservation, 5) conflicts, and 6) liveness. Although these analyses are desirable, FCP simulation of complex PNs is not yet feasible because the execution performance of concurrent logic programs is poor.

A different approach to specification and analysis of PNs is taken by Willson and Krogh. [WILL90] Their goal is to enable a user to specify a system's behavior, to generate models from that specification, and to conduct an efficient and meaningful analysis of the properties of the specification. They describe a rule-based specification language consisting of discrete state variables and state transition rules. They generate, from the specification language, PN models, represented in incidence matrix form, that include timing and stochastic selection choices. They support analysis with a tool that allows a user to specify a submarking of the PN that is of interest and then to perform a reduced reachability analysis. This approach attempts to enable exhaustive analysis by allowing the user to specify those behaviors that are critical.

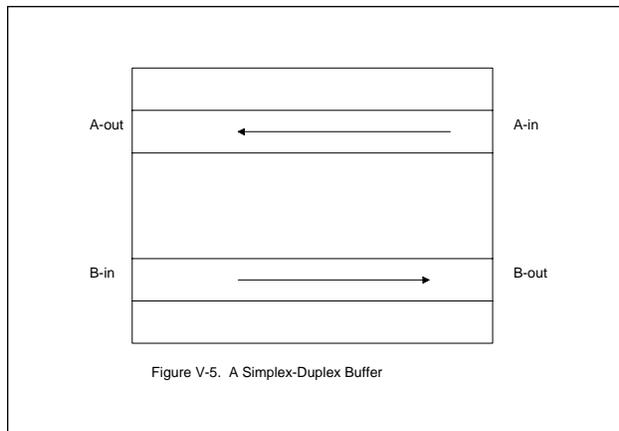
Some recent work attempts to marry PNs and simulation models. Taqi, et al., illustrate techniques for converting SLAM (Simulation Language for Alternative Modeling) models to PNs and vice versa. [TAQI92] These techniques aim at two objectives: 1)

providing an automated means of analyzing PNs and 2) devising a method to pictorially represent complex SLAM models. To accomplish objective one, PN models are translated into SLAM nodes for create, queue, activity, and terminate functions. To accomplish objective two, PNs composed of a very restricted set of representations can be mapped to their corresponding SLAM nodes.

A similar approach is reported by Sakthivel and Agrawal, but for GPSS (General Purpose Simulation System) models. [SAKT92] Here, the simulation blocks permitted include generate, terminate, and seize. In addition, the translation is only one-way, from PN to GPSS.

To close the discussion of Petri net models, the strength's and weaknesses of PNs are recapitulated. The very general nature of PNs is a major source of strength and weakness. PNs are based on a strong mathematical footing that enables a range of analyses. PNs can also

represent many system behaviors, including concurrency, synchronization, non-determinism, timing, and function. Unfortunately, PNs entail substantial complexity when representing complex systems. And, when complex



systems are modeled, the analyses that are theoretically possible with PNs become computationally infeasible. Complex PNs also weaken the graphical clarity that normally attends a PN model. To enable PNs to increase their expressive power, and to incorporate time, many different approaches are proposed by researchers. This results in difficulties for practitioners who wish to apply PNs to specific problems. No particular PN extension is better necessarily than another, and thus a

practitioner must carefully consider the purpose of his modeling against the capabilities of various proposed PN extensions and also against the set of tools available to assist in analyzing or exercising PNs that incorporate extensions. Although invented in 1962, PNs remain immature as a means of specifying system behavior.

C. Temporal Ordering

Temporal ordering describes a behavior by specifying valid sequences of actions that occur in response to external events. [ISO87] This form of specifying behavior has its roots in Milner's calculus of communicating systems (CCS). Behaviors can involve choice (both deterministic and non-deterministic)

```

process simplex-duplex-buffer[in-a, in-b, out-a, out-b] :=
  in-a; (in-b; (out-a; out-b; stop
            [] out-b; out-a; stop)
        [] out-a; in-b; out-b; stop)
        [] in-b; (in-a; (out-b; out-a; stop
            [] out-a; out-b; stop)
        [] out-b; in-a; out-a; stop)
end process

```

(a) Temporal Ordering Using Choice Of Sequences

```

process simplex-duplex-buffer[in-a, in-b, out-a, out-b] :=
  one_time_buffer[in-a, out-a]
  ||| one_time_buffer[in-b, out-b]
where
  process one_time_buffer[in, out] :=
    in; out; stop
  end process
end process

```

(b) Temporal Ordering Using Process Encapsulation And Parallelism

Figure V-6. Temporal Ordering Specifications For A Simplex-

between alternate sequences of actions, parallelism among multiple sequences, and synchronization between sequences. To enable repetitive event/behavior patterns to be represented concisely, named processes can encapsulate sequences. To better explain this approach, an example is used.

Figure V-5 depicts a simplex-duplex-buffer that can accept a message on two simplex input channels (A-in and B-in) and will copy the message on the corresponding output channel (A-out and B-out). The input messages can occur in any order and the output is in no prescribed order.

One means to specify, using temporal ordering, the required behavior of the buffer in Figure V-5 is to describe the allowed possible sequences of events, as shown in Figure V-6 (a). The process is named `simplex-duplex-buffer` and has four parameters - one for each port. The `;` and `[]` operators denote sequence and choice, respectively. As the reader can easily see, either event `in-a` or `in-b` can occur first. Listed after each event is the sequence of actions/events that can occur after the initial event occurs.

Figure V-6 (b) shows an alternative specification of the same behaviors, but relying on process encapsulation (nesting one process within another) and the parallel operator, `|||`. Here, a process, `one_time_buffer`, is defined to model the sequential behavior of one channel and then this process is instantiated twice, once for the a-channel and once for the b-channel. The instantiations then operate in parallel. The described behavior is the same as for the explicit specification given in Figure V-6 (a).

To increase the specification power of temporal ordering a number of operators are usually allowed.² In addition to

² Here, temporal ordering draws on the specific implementation known as LOTOS. [ISO87] LOTOS is perhaps the most ambitious and advanced language for writing temporal ordering specifications. LOTOS is considered specifically in section VI, Languages For Designers.

sequence of actions (;), choice of sequences ([|]), and parallelism among processes (|||), temporal ordering can support synchronization among subprocesses (||), sequential composition of processes (>>), and process disabling ([>). For at least one temporal ordering language, LOTOS, a graphical notation, G-LOTOS, has also been defined. [ISO92]

One use of temporal ordering specifically targets the Ada language. [ROSE91] Rosenblum describes a task sequencing language (TSL) that allows a user to specify acceptable task sequencing at a high level of abstraction and then to annotate Ada programs with TSL statements that embody the specification. Once proper TSL statements are embedded as comments in an Ada program, a set of compile-time and run-time tools can be used to monitor program behavior for conformance with the specification. During run-time, an Ada program, when properly instrumented, outputs significant specification events to a user-controlled monitor. The monitor compares the sequence of events received with the specification of allowable sequences.

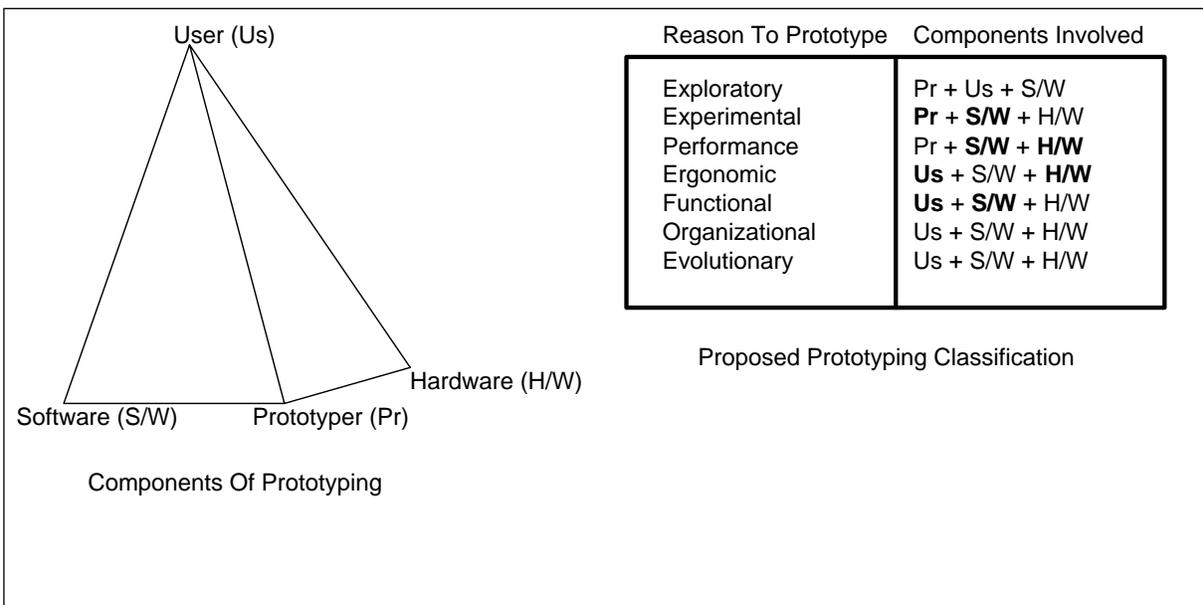


Figure V-7. A Classification Of Prototyping By Reason And Components [MAYH87]

Using TSL with Ada programs comes with a set of problems that apply to most temporal ordering approaches. First, the set of allowed sequences is difficult to specify. The run-time monitoring will not detect specification errors; only differences between the specification and actual run-time behavior can be detected. Second, a running system typically generates many sequences of events; merging events into proper order is difficult and, in fact, cannot be accomplished flawlessly from outside of the run-time system. Third, temporal ordering, as the name implies, captures only the relative ordering among events. Temporal ordering cannot deal with timing constraints, e.g., event A must occur with 3 ms of event B.

D. Modeling and Simulation

"During the past few years there has been an ever-increasing awareness that a static paper description of a computer-based information systems, however formally specified or rigorously defined, is far from adequate for communicating the dynamics of the situation." [MAYH87, p. 481] "Predicting the behavior of real-time applications, particularly in abnormal situations, gets more difficult as the applications become more complex." [HARD88, p. 48] "Because of the size of many real systems, simulation and prototyping may be the only practical forms of analysis." [CAME91, p. 562] For these reasons, system developers are turning, more often than in past years, to the construction of prototype and simulation models to animate system specifications and designs. [BROW88] In fact, although prototypes and simulation models are traditionally treated as separate tools for addressing different problems, some recent work proposes that simulation models and various forms of prototypes should be viewed as part of an integrated toolbox of approaches for exploring a system's characteristics. Figure V-7 illustrates this view.

The prototyping classification in Figure V-7, due to Mayhew and Dearnley, nicely captures several aspects of prototyping. First, prototyping involves various components including the user, the prototyper (e.g., the designer or analyst), the software, and the hardware. Second, prototyping can be motivated by different reasons. Depending on the reason for building a prototype, different components will be involved and some, shown in Figure V-7 in bold typeface, may be emphasized. The classifications of direct interest in the present paper involve the prototyper and the software. Those classifications include exploratory, experimental, and performance. The purpose of exploratory modeling is to elicit and refine the logical requirements of the system. Experimental prototyping encompasses exercising essential aspects of or alternate proposals for the system design. Performance modeling is a special case of experimental prototyping with emphasis placed on evaluating the system under load. In the paragraphs that follow, a variety of approaches to specification and design modeling are considered.

E. Executable Specifications

One method of system modeling entails describing a system's requirements in a formal specification language and then exercising the specification to assess various interesting properties. Several researchers have proposed languages and run-time environments for modeling system specifications. The present paper considers those proposals intended for distributed and real-time systems.

Zave developed a Process-oriented, Application and Interpretable Specification Language (PAISLey) intended to validate the feasibility of requirements and to act as an executable design. [ZAVE82, ZAVE86] PAISLey merges asynchronous processes with functional programming processes represented as finite state machines. Inter-process communication is handled

via exchange functions that model a rendezvous. As described in 1986, PAISLey possessed seven significant features. First, PAISLey allowed modeling of maximal parallelism between processes. The only restriction on parallelism requires that a process, internally, must be synchronized at the end of each process step. No other language, at the time, allowed both synchronous and asynchronous parallelism free from concern with mutual exclusion.³ A second significant feature of PAISLey is encapsulated computation, i.e., every action, except for inter-process exchanges, in a PAISLey specification is a mathematical function. Another useful feature is the tolerance of incompleteness. The PAISLey run-time can choose among a possible set of function evaluations when none is explicitly defined. The run-time system can also query the user for the missing evaluations. A fourth feature of value is PAISLey's ability to evaluate timing constraints. Any function can be augmented with a time variable denoting an upper or lower bound, a distribution, or all three. The interpreter then honors timing constraints where possible and reports failures. The specified timing constraints are combined with a model of system overhead to enable a specification to be assessed for performance. The PAISLey interpreter also ensures, when the specifier restricts use of recursion, that specifications can be executed within a bounded space and time. No process can be starved by the interpreter because every event is executed on a FIFO basis.

A sixth significant feature of PAISLey is consistency checking. Of course, many of the conditions that cause undefined program states during execution cannot occur in PAISLey specifications because the language and interpreter are

³ Functional languages have no asynchronous processes. Languages such as CSP represent processes as sequential. Languages such as Ada allow shared variables and thus face problems with mutually exclusive access.

defined to avoid or account for such conditions. PAISLey can, however, check for timing constraints and for system deadlock. A final feature attributed to PAISLey is ease of specification. The syntax includes set expressions (using only three operators), mapping expressions (using three combining forms), timing constraints, and a single, replication notation.

For all its significant features, PAISLey also exhibits some interesting shortcomings. For example, PAISLey specifications are operational, specifying how, not what. This means that users must specify a system with too much precision. If one chooses to view PAISLey as a means to execute designs, then the inefficiency of the interpreter becomes a problem. In summary, PAISLey models fall somewhere between a requirements and design specification. The result is largely unsatisfactory for both purposes.

Lee and Sluzier describe an executable language, SXL, for modeling simple behavior that aims directly at modeling requirements. [LEE91] SXL encompasses a state transition language. Each model may include invariants and each transition in a model has associated pre- and post-conditions. The invariants and other constraints are expressed with a combination of entity-relationship (E-R) structures and quantified, first-order logic. The finite state machine (FSM) interpreter underlying SXL is implemented in Prolog. SXL cannot model parallel systems because each specification consists of a single FSM. Using SXL an analyst builds a specification by: 1) deriving an E-R model of the requirements, 2) expressing the model as SXL objects and facts, and 3) mapping transitions from an informal requirements description to SXL events, transitions, and constraints. The most significant benefit from using SXL, as reported by Lee and Sluzier, is that, while building an SXL model, incomplete, inconsistent, and ambiguous requirements are often uncovered.

A recently devised specification language, L.0, targets descriptions of protocols and similar reactive systems. [CAME91] L.0 is a rule-based system (where rules can be activated and deactivated dynamically and several rules may fire simultaneously), that includes encapsulation, data sharing, indirection, quantification, and recursive definition. L.0 rules are of two forms: cause-effect and constraints. Cause-effect rules provide three general semantics: 1) once <event> then <effect>, 2) until <event> then <effect>, and 3) whenever <event> then <effect>. Constraint rules simply capture invariants, using a maintain <predicate> syntax. L.0 modules comprise named rule-sets that can be suspended, resumed, removed, and activated. Parallelism among rules and modules is permitted, as well as a limited degree of non-determinism. For a simple protocol specification, an L.0 rule-space contains between 300 and 400 cause-effect rules. On average, 3% of these rules are triggered at each program step. L.0 supports simulation and prototyping because state explosion within protocol specifications makes verification a difficult problem.

The executable specification approaches covered thus far require the analyst to learn the syntax and semantics of an unfamiliar language. A different approach, described by Harding, uses a set of computer-aided software engineering (CASE) tools, under the name Foresight, to model specifications of embedded systems. [HARD88] The CASE tools include graphic editors, supporting the notation from structured analysis and design technique (SADT) with real-time extensions, that allow an analyst to create two models. The functional model describes the basic system logical operation. The constraint model specifies time-critical relations between the system and external events. The CASE environment includes tools for generating executable models, including models of both hardware and software, from static specifications and then to assess the

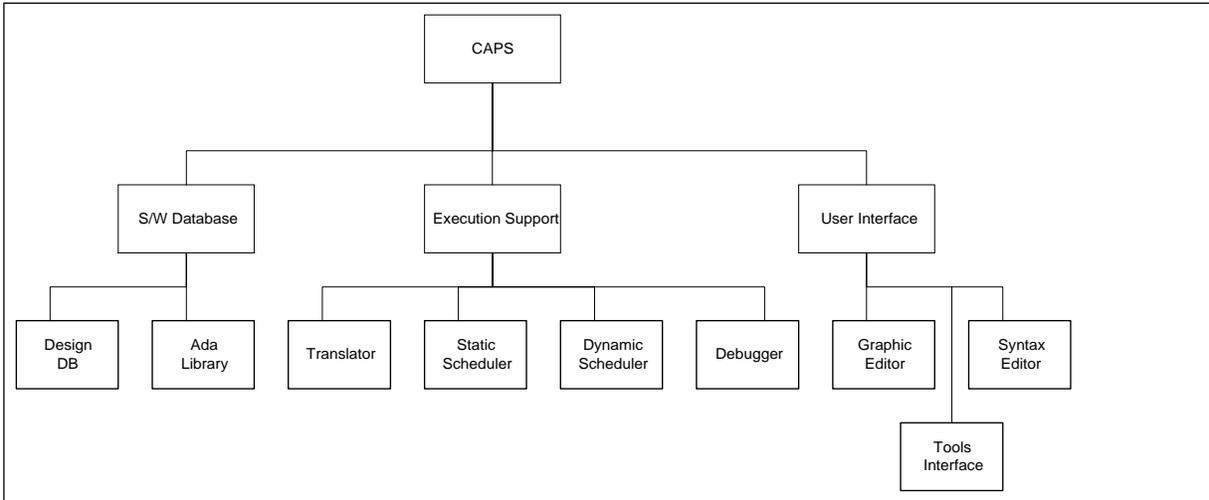


Figure V-8. Components Of The CAPS Environment [LUQI92]

performance of the system. By relying on SADT notation, Foresight sacrifices the precise semantics available with other languages, but gains a user interface that most analysts find familiar.

One final approach to executable specifications deserves mention because of its uniqueness. Nota and Pacini view the inspection of software behavior as a process of querying executable specifications. [NOTA92] Using queries, an analyst can isolate the subclass of possible behaviors to a critical set that might possibly be subjected to an exhaustive analysis. Nota and Pacini define a query language, RSQ, that allows analysts to construct queries against executable specifications that are expressed in a language called RSF. This approach is similar to selecting a reduced reachability graph for a Petri net.

An alternative to using executable specifications is to transform specifications into prototypes via a translation. Transformable specifications are discussed next.

F. Transformable Specifications

Transformable specifications typically enable an analyst to describe the essential characteristics of a system design in a

very high-level language that can subsequently be translated into an executable system. The executable system usually consists of modules coded in a high-level programming language. Some of the executable modules are generated from the high-level specification, while others are extracted from a library of commonly used components. Three examples are described below.

Luqi describes a computer-assisted prototyping system (CAPS) for generating a color, multi-window command and control application. [LUQI92] The generated prototype consists of Ada code. The intent of CAPS prototypes is threefold: 1) to evaluate the structure and performance of a proposed design, 2) to refine the system requirements, and 3) to assess the feasibility of the functional specification. CAPS encompasses a number of components as shown in Figure V-8.

Designers specify, using the Prototype System Description Language (PDSL), the following elements: 1) functions, 2) data streams (that link functions together), 3) maximum function response times, 4) function triggers, 5) function output messages, 6) a reference to the system requirements specification, and 7) execution time estimates for each function. Using the provided information, the CAPS static scheduler generates a feasible schedule (if one exists) for a cyclic executive. The CAPS translator generates Ada code and then binds together the generated code with any needed components from the CAPS Ada library. The CAPS dynamic scheduler is used to allocate any excess time (i.e., time not required to meet the static schedule) to non-critical system functions. The CAPS debugger monitors the system constraints as the prototype executes and enables the designer to make adjustments while the system is running. To construct a prototype, the designer typically uses the steps shown in Table V-3.

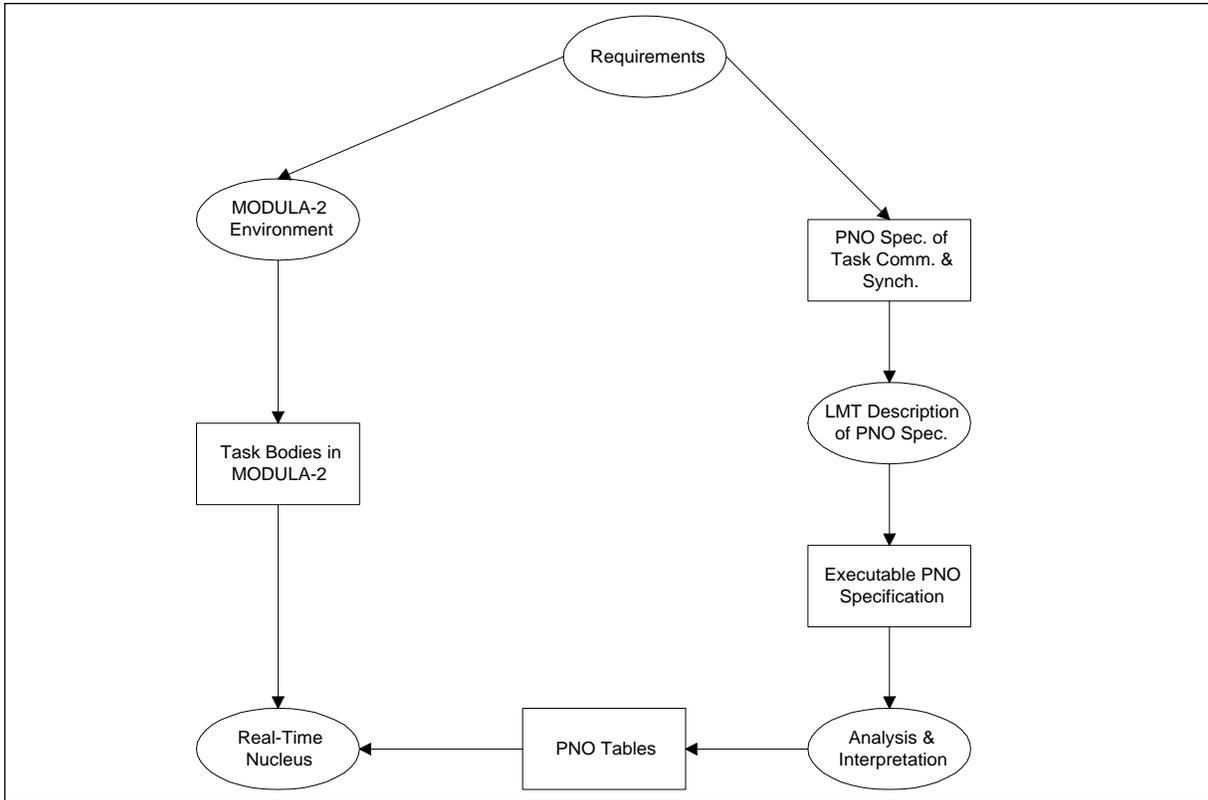


Figure V-9. Prototyping With HLPNs And MODULA-2

Although all of the CAPS functions illustrated in Figure V-8 have not yet been implemented, CAPS has successfully produced Ada prototypes of command and control systems. The prototypes were produced quickly and with low cost. [LUQI92] Some shortcomings of CAPS are also reported. CAPS does not address distributed systems because three issues remain unsolved: 1) no method exists to evaluate global timing constraints (such a method is necessary to generate complementary schedulers among multiple nodes), 2) no method exists to bound the delivery times on messages exchanged between nodes, and 3) no methods exist to detect or prevent deadlocks between nodes.

A different approach to generating prototypes, described by Sahraoui and Ould-Kaddour, proposes writing sequential tasks in Modula-2 and describing task interactions with an extended Petri

net model, called Petri nets with objects (PNO). [SAHR92] The PNO model is supported with a language, LMT, that allows PNO

Table V-3. Producing A Prototype With CAPS

1	Designer draws the system computation graphs (i.e., DFDs).
2	CAPS editor generates skeleton PDSL code.
3	Designer modifies PDSL skeletons to produce a prototype description.
4	CAPS translator produces Ada packages that instantiate data streams, system reads and writes, and function executions. Interfaces to the static scheduler are also generated.
5	CAPS static scheduler searches for a feasible schedule and, if found, generates an Ada package with the static schedule represented as a task.
6	CAPS dynamic scheduler produces an Ada package encapsulating a dynamic schedule for non-critical functions.
7	Designer writes any necessary Ada code that is not available in the CAPS Ada library.
8	CAPS compiles the Ada code and then loads the system and starts execution.
9	System users observe and evaluate the prototype results.
10	Designer modifies the prototype as necessary.
11	Once the prototype behavior is acceptable, the code is optimized and ported to the target system.

specifications to be translated into an executable form for analysis and interpretation. The PNO model replaces PN tokens with objects that possess a semantic meaning. When a transition fires an object is removed from the incoming place and an object is produced at the outgoing place. For modeling multitasking systems, PNO transitions represent a precondition and an associated action, tokens portray messages, and places model tasks (written in Modula-2), mailboxes, and synchronization points. Figure V-9 provides an idea of how the Modula-2 and PNO/LMT environments are integrated.

Another approach to transformational prototyping involves translating temporal ordering specifications (in LOTOS, see section VI), into C functions which are then executed by cooperating processes in UNIX. [VALE93] Each LOTOS process definition is translated into an extended FSM.⁴ The multi-way rendezvous included in the LOTOS language is implemented via an algorithm based on inter-process message passing. No support is provided for translating LOTOS abstract data types. (See later parts of section V and see also section VI for information on abstract data types and LOTOS.) To build prototypes using this method, LOTOS specifications must be free from unbounded recursions.

A hybrid approach to prototyping with transformational specifications is advocated by Choppy and Kaplan. [CHOP90] They propose a method for incremental development of large, modular software systems. Modules comprising a system may interact even when the modules exist at different states of development. Each module may be fully abstract (existing solely as an algebraic specification), may be fully concrete (implemented in a programming language), or at a mix of points between abstraction and concreteness. They define an algebraic language (PLUSS) through which axioms can be constructed as Horn clauses built over equations or predicates. They also describe an execution environment, ASSPEGIQUE, that can perform mixed evaluation of Horn clauses augmented with concrete implementations. The concrete portions are implemented in Ada.

G. Testbed-Based Prototyping

⁴ This reveals an interesting relationship between temporal ordering and finite state automata (FSA). Temporal ordering specifications describe allowable behaviors but provide no clue as to generating a system that exhibits such behaviors. Systems that behave according to an extended FSA are easy to generate but verifying that an observable sequence of external events conforms to a given extended FSA remains a difficult problem.

Chu, et al., advise that prototypes can be used to best advantage when experimental implementations are exercised in testbed environments. "Testbeds can be configured to represent the operating environments and input scenarios more accurately than software simulation. Therefore, testbed-based evaluation provides more accurate results than simulation and yields greater insights into the characteristics and limitations of proposed concepts." [CHU87] Chu describes two tightly-coupled, multi-computer testbeds that provide efficient inter-node communication and full connectivity among processors and memory. The testbeds can support the validation of design techniques for distributed, real-time systems. Chu reports on using the testbeds to study the behavior of: 1) distributed algorithms, 2) recovery schemes, 3) distributed database locking techniques, and 4) update strategies for replicated data. Testbeds can provide realistic modeling of distribution; however, constructing and maintaining testbeds of sufficient flexibility can be expensive. In addition, the construction of prototypes in testbeds can also prove labor-intensive.

H. Simulation

Simulation is a form of prototyping particularly appropriate for system performance evaluation. "Simulation is the process of designing a model of a real system and conducting experiments with this model with the purpose of either understanding the behavior of the system or of evaluating various strategies...for the operation of the system." [ZEIG84, p. 2] Simulation presents an analyst with three difficult problems: 1) choosing a level of detail in the model compatible with the analyst's modeling objectives, 2) verifying that the model accurately represents the modeled behavior, and 3) validating that the model reflects the behavior of interest.

In general, simulation models can be constructed using three approaches. The most widely known approach requires an

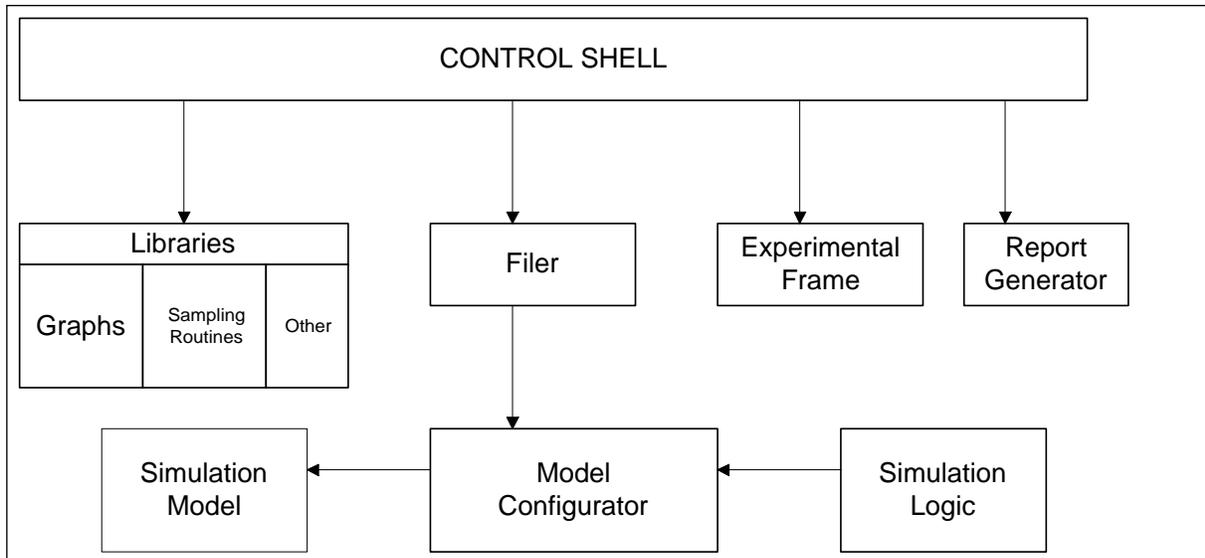


Figure V-10. General Structure Of A Simulation Generator
[PIDD92]

analyst to construct an abstract model, to identify the salient model parameters and values of interest, to select an experimental methodology and metrics, and then to code the model in a simulation language. [ZEIG84] This approach is often used to assess the performance of communications protocols and to evaluate various communication network configurations.

For example, Finn, et al., simulated the design of a hierarchical system of multiple access busses in a real-time control system. [FINN92] They wished to estimate, within a specific probability, the delay of two types of messages, one with a maximum delay constraint of 1 ms and one with a constraint of 1 second, under expected loads, given a specific configuration of nodes connected by busses with a maximum capacity of 1 Mbps each. Initially, they performed a mathematical analysis. The results of the analysis were suspect because a number of restrictive assumptions were required to keep the model tractable. Next, they constructed a functional simulation using Pascal. The Pascal model proved efficient and flexible, but lacked a graphical user interface and was also difficult to debug. Finally, they use a simulation tool, the

Block-Oriented Network Simulation (BONeS), which proved useful for determining the details of hierarchical network behavior, bus interface delays, and actual maximum queue depths. Unfortunately, the detailed BONeS model executed very slowly. The three, different methods described by Finn illustrate some of the tradeoffs that must be considered when using a simulation model.

Parr and Bielkowicz, too, resorted to simulation to evaluate the performance and behavior of a communication system. In particular, they proposed a new, self-stabilizing, bridge protocol (to replace the IEEE 802.1D spanning tree algorithm) for interconnected ethernet. [PARR92] They also found analytical models to require unrealistic assumptions. In addition, they pointed out that analytical models can only capture steady-state behavior and, therefore, cannot evaluate a system's behavior under transient conditions. Both Finn and Parr found analytical models to be a useful tool for verifying more detailed simulation models.

Because building, verifying, and validating simulation models require great skills and incur high expense, researchers are investigating methods to generate simulations from libraries of generic, domain-specific models. [DEME91, OZDE93, PIDD92, ZEIG87] Figure V-10 illustrates the general components of a system to support model generation. A number of general-purpose, data-driven simulators have been developed, including GPSS, HOCUS, and WITNESS. Domain-specific, data-driven simulators include: SIMFACTORY (written in SIMSCRIPT II.5), MAST (written in FORTRAN), PROPHET, and XCELL+.

Ozdemirel and Mackulak describe an approach that allows users to construct specific models of manufacturing systems by selecting and then configuring a pre-built, generic model. [OZDE93] In their system, 14 generic model modules were written using 2500 lines of SIMAN code. A user interface, composed of

8000 lines of Turbo Prolog code, acts as an expert adviser for model selection and enables a user to configure the model. They propose their approach based on the belief that the most difficult skill required of a simulation designer is development of a conceptual model. Their approach reduces conceptual model development to an expert system-assisted search.

DeMeter and Deisenroth propose a framework for construction of heterogeneous models for simulating multi-stage manufacturing systems. [DEME91] Heterogeneous models consist of a mix of highly detailed models interspersed among a larger set of generalized models of low detail. This enables modeling of specific parts of a system, or design, in enough detail to

assess behavior, while allowing the model to also be observed within a simulated environment. This approach is particularly suitable for evaluating new communications protocols operating under a simulated, network load.

In summary, simulation models can prove useful for assessing both the behavior and performance of distributed, real-time systems. Properly constructed models, augmented with accurate parameters and effectively designed experiments, can be used to assess a

<p>TYPES STACK[X]</p> <p>FUNCTIONS <i>empty</i>: STACK[X] -> BOOLEAN <i>new</i>: -> STACK[X] <i>push</i>: X x STACK[X] -> STACK[X] <i>pop</i>: STACK[X] - -> STACK[X] <i>top</i>: STACK[X] - -> X</p> <p>PRECONDITIONS pre <i>pop</i> (s: STACK[X]) = (not <i>empty</i>(s)) pre <i>top</i> (s: STACK[X]) = (not <i>empty</i>(s))</p> <p>AXIOMS For all x: X, s: STACK[X]: <i>empty</i>(<i>new</i>()) not <i>empty</i> (<i>push</i>(x, s)) <i>top</i> (<i>push</i>(x,s)) = x <i>pop</i> (<i>push</i>(x,s)) = s</p>

Figure V-11. Example ADT For A Stack [MEYE88, p. 55]

system's typical performance under steady load, worst-case performance under peak load, and response to transient conditions. Unfortunately, a simulation model is only as effective as it is accurate. Model builders need skills in: 1) conceptual model development, 2) translating from conceptual model to executable model, 3) analysis for estimating model parameters, 4) experiment design, and 5) analysis of model results. In addition, modelers need an understanding of the problem domain and specific system to be modeled. Individuals possessing such skills can be found only rarely. Even when such experts exist, extensive time and effort are involved in building a simulation, verifying and validating the model, designing and conducting experiments, and then interpreting the results. Time and effort translate into expense.

This completes consideration of formal models and methods for specifying and analyzing system behavior. The final three formal methods discussed, temporal logic, axiomatic methods, and abstract data types, are structural models.

I. Abstract Data Types

Abstract data types (ADTs) encompass a means and a theory for specifying mathematically the essential characteristics of a data type, or class. An ADT specifies the name of a data type, the functions available to manipulate the data type, and a set of axioms that characterize the data type. Some of the axioms of an ADT, so-called invariants, describe properties that will always hold. Other ADT axioms specify the preconditions that must hold for a specific function before the result can be obtained. ADTs can be specified using first-order, quantified logic (FOQL) or using an algebraic notation. Figure V-11 gives an example ADT for a stack specified using FOQL. (See section VI, LOTOS, for an example ADT specified algebraically.)

The stack ADT in Figure V-11 consists of four sections. **TYPES** specifies the name of the ADT, **STACK**, and indicates

elements of any type, X , can be placed on the stack. **FUNCTIONS** contains the syntax of the operations provided by STACK; the syntax includes a function name (shown in italics), any input parameters, a function arrow (\rightarrow denotes a total function and $\dashv\rightarrow$ denotes a partial function), and any results. Total functions will achieve the indicated result under any input conditions. Partial functions can achieve the stated result only under restricted input conditions, so for each partial function a precondition must be given that specifies the conditions under which the associated function will achieve the intended result. In the example, functions *pop* and *top* will not work when a stack is empty. The final section of the ADT contains **AXIOMS** defining the semantic properties of the ADT. In the example, the axioms apply for all elements of type X and for all stacks of type STACK[X]. For example, when *top* is called immediately after element x is pushed onto stack s (*push*(x, s)), the element x will always be returned. Each axiom listed will always be true for the ADT STACK[X].

ADTs provide a convenient means for specifying formally the properties of information hiding modules in a software design. ADT specifications are static and require that a program be written to generate the specified behavior. This can present a problem when simulating designs because the program underlying an ADT must be implemented in order to present an active interface. Wang and Parnas are investigating one possible method of animating information hiding modules (IHMs) from module specifications. [WANG93] They propose specifying an IHM using trace specifications. They suspect that, given trace assertions for a trace specification, the externally observable behavior of a module can be simulated through trace rewriting rules. In effect, they view ADTs as finite state machines that can accept inputs and simulate responses using a trace rewriting system. Should Wang and Parnas achieve acceptable results, IHMs

could be specified and simulated within a design without having to implement the underlying, application-specific, code.

In summary, the reader should understand that writing a formal ADT specification is difficult work. Once an ADT is specified, the specification must be animated in some way to support design simulation. In addition, ADTs cannot be used to describe the behavioral or correctness properties of sequential tasks. Axiomatic methods provide more aid in specifying tasks.

J. Axiomatic Methods

Sequential programs comprising constructs for choice, sequence, iteration, assignment statements, and subprogram calls can be specified as a set of axioms using FOQL. In general, the approach requires that a program result be formally specified and then that a set of programming steps be derived that will enable the result to be obtained, given a determined precondition, provided that the program terminates. At each step in the program derivation, appropriate statement preconditions or loop invariants are found. When a program has been completed, proof exists that a program, S , will achieve a known result, R , given a specific precondition, Q . This relationship is usually specified as $\{Q\} S \{R\}$. When $\{Q\} S \{R\}$ holds for every step in a sequential task, the task is said to be partially correct. For concurrent programs, safety must also be ensured. A safe program will never enter an unacceptable state such as conflicting access to shared data, deadlock, critical races, or starvation.

Two means exist to prove programs correct: 1) operational proofs and 2) axiomatic proofs. [KARA91] Operational proofs entail symbolic execution of a specification and then an evaluation of the resulting execution tree. This is similar to program testing. Operational proofs are best used when axiomatic proofs are not possible or not practical. Axiomatic proofs use the rules of a system of logic or algebra to

establish program correctness against a specification. Recent research aims at applying these methods to concurrent programs.

The approach proposed by Dillon requires that each task's intended behavior be axiomatically specified as described above. [DILL90G] Each task is then executed symbolically to generate trees of every possible task state. From the execution trees, a set of predicate logic formulae (verification conditions) are generated. Any program for which these verification conditions can be proved is known to be partially correct.

After all tasks are verified, assertions (consisting of local and global invariants, augmented with auxiliary variables) are inserted into the tasks and a higher level symbolic execution tree is generated to evaluate the safety properties of the concurrent program. To ensure safety, distributed termination of all tasks must be shown and absence of rendezvous failure must be assured. (Since Dillon's method applies to Ada programs, rendezvous failure means that a select statement has no open alternatives or that an entry call is invoked after a task has terminated.)

The work reported by Dillon is limited to Ada programs and addresses only logical correctness and a limited set of safety properties. Extensions are needed to incorporate timing information into the axioms so that the real-time behavior of a program can be expressed and then proved. Other researchers are also investigating this problem. For example, Ravn, et al., propose specifying real-time requirements as formulae in a duration calculus (also called a real-time interval logic) where predicates define the duration of states. [RAVN93] The top level design of a system describes a control law, that is, a finite state machine controlling transitions between phases of an operation. The work of Ravn, et al., combines finite state machines, ADTs, represented with Z (see section VI), and temporal logic.

K. Temporal Logic

Temporal logic can be used to describe sequences of program states (much as temporal ordering). Most temporal logic systems begin with FOQL and then add a set of temporal operators. The most often encountered temporal operators include: 1) eventually, 2) next, 3) until, and 4) henceforth. These extend the ability of logic systems beyond the typical temporal operators: there exist and for every. Temporal logic can be applied to specify and analyze selected properties of concurrent systems.

Karam and Buhr describe the application of temporal logic to analyze concurrent Ada programs for deadlock. [KARA91] They propose a specification language, COL, supported by a specification analyzer written in Prolog. An Ada system, composed of N concurrent, infinitely-executing tasks, is specified as an N-tuple of the control and data states for each task. The system state changes whenever the state of one task changes. Discrete time is modeled, then, as a sequence of system states.

The COL specification language adds the four, typical, temporal operators to FOQL, but also provides a built-in library of predicates specifically for Ada. To simplify the analysis of specified programs, several Ada features are excluded: 1) dynamic task creation and destruction, 2) timed or conditional task calls, 3) delay or else selective accept alternatives, 4) exceptions, and 5) dynamic data creation (this eliminates procedural recursion). While excluding features (1) and (5) above seems acceptable, exclusion of the remaining features might overly restrict the form of Ada programs that can be specified with COL. Still, Karam and Buhr report that the

"...COL language paints a limited, but useful picture of the Ada language." [KARA91, p. 1124]

Temporal logic does extend the specification and reasoning power of FOQL so that time ordering can be considered. Still, as with temporal ordering, specific timing constraints cannot be described and reasoned about. This limitation also holds for ADTs and for axiomatic methods in general. In addition, these methods are difficult to use for specification. Worse, reasoning with these methods is sometimes labor-intensive, always error-prone, and, when automation can be applied, computationally-intensive, sometimes to the point of infeasibility.

The formal methods and models covered in section V, often provide the underlying theory for design and specification languages. Languages strive to enhance the underlying formalisms with some suitable syntax and, usually, with a run-time environment that can help a designer animate proposed designs. In the next section, some design and specification languages, based on the formal methods and models discussed above, are considered.

VI. Languages For Designers

Languages implementing some of the formal models described in section V can help designers to describe and exercise specifications and designs. The following paragraphs discuss some representative design and specification languages: Communicating Sequential Processes, Zed, Communicating Shared Resources, Extended State Transition Language, and Language of Temporal Ordering Specification.

A. Communicating Sequential Processes

Anthony Hoare proposed a mathematical notation and semantics for specifying cooperative behavior between sequential processes that communicate. [HOAR85] The notation, called

```

[producer::
  *[{generate item} -> buffer ! item]
//
buffer::
  [content : (0..n-1) item;
   incount, outcount : integer;
   incount := 0; outcount := 0;

   * [incount < outcount + n; producer ? content
      (incount mod n) -> incount := incount + 1;
      [] outcount < incount; consumer ? request() ->
         consumer ! content (outcount mod n);
         outcount := outcount + 1
      ]
  ]
//
consumer::
  * [buffer ! request(); buffer ? item; {use item}]
]

```

Figure VI-1. CSP Program Of A Buffered Producer-Consumer System [HULL86, p. 501]

Communicating Sequential Processes (CSP), combines first-order logic, set theory, functions, and traces to define a process logic with synchronization based on synchronous message exchanges. (CSP also allows shared data with a limited semantics.) Hoare added, to the syntax and semantics, laws for reasoning about the behavior of processes. CSP goes quite far toward defining a theory of distributed, concurrent systems. Each CSP process is represented as a sequential program (which can terminate) that operates according to program statements that can be both deterministic and non-deterministic. CSP processes interact via messages. Hoare shows how CSP can be used to specify interruptable (with resume) processes, restart after failure, alternation among behaviors, checkpoints, and shared resources. In the main, CSP aims to detect or avoid deadlock, starvation, and livelock in concurrent systems.

Hoare chose to reject certain features so that CSP could remain simple and clear. For example, shared-storage is not

supported, nor is multi-threading within processes. These omissions eliminate such models as conditional critical regions, monitors, and nested monitors. Hoare finds that Ada is well designed (if quite complex) for multiprocessor implementations using shared data, so he chose to emphasize distributed processes. Regarding the controversial area of communication paradigms, Hoare prefers an RPC model, limiting inter-process message exchange to synchronous communications. He considered and rejected single and multiple, buffered channels, bi-directional buffered channels, functional multiprocessing, and unbuffered communications.

A number of researchers started with CSP as a base for a multiprocessing language. In each case, CSP could not be used without change. [HULL86] The CSP inter-process communications paradigm proved most troubling. CSP processes communicate, and synchronize, with input and output commands. The general form is **source?variable** for input and **destination!variable** for output. CSP provides guarded alternative and repetition constructs to enable multiple, iterative message reception. A sample CSP program is shown in Figure VI-1.

Since all CSP communications is tightly-coupled, loosely-coupled communications can only be simulated by introducing an intermediate process (a common occurrence with Ada), as shown with the buffer process placed between the producer and consumer processes in Figure VI-1. In CSP alternating behavior is denoted by `*[...]`, choice by `[..[]..]`, parallelism by `//`, sequence by `;`, and guards are followed by `->`. Statements enclosed in `{}` are comments. The example in Figure VI-1 can probably be followed without further explanation.

Each CSP program is specific to the names of the destination and source processes that make up the program. This proves most unsatisfactory when writing processes that must be used in a variety of systems. Another shortcoming of pure CSP

is the allowance for non-determinism. Non-determinism in programs is usually only acceptable when the guards that are enabled simultaneously have equal priority. In other cases, some order of selection must be imposed on the guarded statements. A final drawback of CSP is the lack of support for data types.

Researchers at the University of Adelaide implemented CSP as COSPOL. [HULL86] COSPOL adds asynchronous communications to CSP and includes Pascal data typing. In addition, non-determinism is restricted to guarded alternative statements used for message input. Another implementation, CSP/80, was the product of a group of academics in the UK. [HULL86] CSP/80 adds the concept of a communications port to CSP; thus, CSP/80 processes are de-coupled from the identity of the processes with which they communicate. CSP/80 allows C data types, but with strong typing. CSP/80 supports modularity and also allows output statements within guards (a feature not permitted by CSP). CSP/80 does support the full non-determinism of CSP. Perhaps the most famous implementation of CSP is known as Occam, a low-level language developed by Inmos, Ltd. for the Transputer. [HULL86] One can view Occam as an assembly language for CSP. Occam, an untyped language, provides basic statements for sequence, parallelism, choice, and while loops. All Occam inter-process communication is via unbuffered, unidirectional channels. Occam allows both determinism and non-determinism in choice statements. Occam, as with CSP, does not permit output statements in guards. Of the three implementations reported here, Occam aligns most closely with CSP. The only real enhancement provided by Occam is the introduction of channels to de-couple processes from the names of other processes.

Later, Brinch Hansen used CSP and Pascal to form the basis of a distributed systems programming language he called Joyce. [HANS87] Joyce permits processes to exchange messages via

$[Location, Value]$

| $bound : N$

Sensors

$readings : Location^+ \rightarrow Value$

$areas : PLocation$

$\#areas \leq bound$

$dom\ readings \subseteq areas$

Update

$\Delta Sensors$

$l? : Location$

$v? : Value$

$l? \in areas$

$readings' = readings \oplus \{l? \rightarrow v?\}$

$areas' = areas$

Figure VI-2. Sample Zed Specification Of A Simple Sensor ADT

synchronous, bi-directional channels that may be shared by two or more processes. A Joyce rendezvous, however, always involves exactly two processes. When more than two processes are ready to rendezvous on a channel, two are selected arbitrarily. Joyce allows processes and channels to be created dynamically. Processes can also be activated recursively. Because Joyce uses Pascal for data typing, messages exchanged between processes can be of different types, even across the same channel. This permits the Joyce compiler to check message types.

Although CSP and the languages that implemented CSP never achieved a large, practical presence in the marketplace, they did influence the thinking of designers of later languages. The reader will perhaps be able to detect some of these influences when Estelle and LOTOS are discussed later in this section.

B. Zed

Zed is a language for specifying ADTs and systems of ADTs. [POTT91] Zed, initially devised at Oxford University's Programming Research Group, is based upon first-order logic and special set theory. [DILL91D] Zed uses a familiar two-valued system of logic (as opposed the Vienna Development Method which uses a three-valued logic). Zed has been used at IBM to re-specify the Customer Information Control System (CICS). Re-specifying CICS in Zed enabled IBM analysts to discover a number of errors and omissions that had not be detected even though CICS is a twenty-year-old commercial product.

Zed specifications yield a functional description of what a system is to do, as opposed to how a system is suppose to accomplish its objectives. This declarative approach, sometimes called operational abstraction, leads to concise, unambiguous, exact specifications that are easy to reason about. Zed also employs representational abstraction by using high-level, mathematical concepts without worrying about how these concepts will be implemented.

The main syntactic tool of a Zed specification is known as the schema. Each Zed schema contains a schema name, a set of definitions, and a specification of the post-conditions associated with any preconditions required by the schema. In general, Zed schemas specify one operation in an ADT or system. Figure VI-2 gives an example of using Zed to specify a simple, but incomplete, sensor ADT.

The main schema, named *Sensors*, comprises a partial function, *readings*, that maps from a *Location* to a *Value* (*Location* and *Value* are defined as sets). The set *areas* is defined as the power set of the set *Location*. The variable *bound* is a schema constant from the set of positive numbers. *Sensors* defines two invariants: 1) the number of *areas* cannot

exceed the *bound* and 2) the domain of *readings* must be a proper subset of *areas*.

```
process Sensor
  local sample
  output data
  timevar t
  every 6 do
    exec(sample);
    scope do idle interrupt send(data) -> skip
      timeout t hard -> skip od
  od

process Conv
  input data
  local compute
  output coord
  loop do
    recv(data);
    scope do exec(compute); send(coord)
      timeout 2 hard -> skip od
  od
```

Figure VI-3. Sample CSR Description Of A Sensor And Converter
[KERB92, p. 772]

The schema *Update* represents an operation in the Sensor ADT. The operation alters the *Sensors* schema. *Update* requires two inputs: *l* is a member of the set *Location* and *v* is a member of the set *Value*. As a precondition to the *Update* operation, *l*, must be an element of the set *areas*. If the precondition is satisfied, then the function *readings* will be updated so that the old value associated with input *Location*, *l*, will be replaced by the new input *Value*, *v*. The *areas* set will not be changed.

In summary, Zed provides a rich set of operators combining first-order logic with special set theory. The notation is, perhaps, too rich for easy use. Zed allows precise and concise specification of the semantics of a system. From a Zed specification additional properties can be reasoned about a

system. Zed provides no clue as to how a operation is to be accomplished.

C. Communicating Shared Resources

Gerber and Lee propose a layered approach to specifying and verifying real-time systems. [GERB92] Their top layer is an application language that allows the specification of time-outs, deadlines, periodic processes, interrupts, and exception handling. Their middle layer comprises a configuration language that can be used to map processes to system resources and to describe the communications links between processing nodes. The application and configuration languages, taken together, compose a specification language called Communicating Shared Resources (CSR). The configuration mapping can be translated into a process algebra, called calculus of CSR (CCSR). CCSR defines a semantics upon which a reachability analyzer is based. The objective of the CSR paradigm is to facilitate the specification of real-time processes and then to enable a static evaluation of various design alternatives. Evaluating alternative designs involves mapping a functional description to various configuration descriptions and running the reachability analyzer on each configuration.

The CSR application language comprises declarations and statements. Declarations include ports for sending output messages and receiving input messages, events for executing local operations, and timing parameters that are used in certain types of statements. CSR application language statements include send and receive, time-outs, periodic loops, interrupts, exception handling, and sequential composition. The emphasis is on describing inter-task operations. Discussing a small example should prove instructive.

Figure VI-3 shows a brief specification of a sensor and converter in the CSR application language. The CSR keywords are rendered in boldface type. The process Sensor contains three

declarations: a local operation (sample), an output channel (data), and a free time variable (t) that can be set from a configuration description. Sensor wakes every six seconds, executes sample and then attempts to output on data. If the output is not accepted within time t, then Sensor simply stops trying. The process Conv loops forever. First, Conv waits for input on data. Once input arrives, the local operation compute is performed and then an attempt is made to send a message on coord. If the message is not accepted in 2 time units, then Conv simply returns to the top of its loop.

CSR provides for three forms of concurrency. Processes can execute concurrently on the same resource and on different resources (i.e., be modeled as a distributed system). The third form of concurrency, intra-process concurrency, can be modeled by the analyst using the **interleave** statement.

The CSR configuration language enables an analyst to declare system resources (**resource**), to bind priority and time values to processes (**process**), to map processes to resources (**assign**), to create channels by connecting ports (**connect**), and to define limits to resources (**close**). The configuration language allows hierarchical schemas for added convenience.

The calculus of CSR defines an underlying semantics using set theory and two sets of inference rules: 1) an unconstrained transition system and 2) a transition system to model preemption and priority. A translator can map the CSR processes into CCSR terms. At first, Gerber and Lee planned to implement the semantic model as a rule rewriting system but using the "...rewrite rules stretches the range of both endurance and patience." [GERB92, p. 781]. They decided to try reachability analysis instead. A CSR specification, after translation into CCSR, is guaranteed to produce a finite reachability graph. Once the system's state-space is generated, real-time errors can be found directly.

A CSR application lacks the abstraction usually found in a requirements specification but a program can be easily produced from a CSR description. CSR seems to be more appropriate as a design tool that as a specification aid. In fact, a underlying model can probably be developed to simulate a CSR application and configuration. CSR seems to hold some promise as a tool for designing and evaluating distributed, real-time systems.

D. Extended State Transition Language (Estelle)

Estelle is a language for describing formally the properties of communications protocols and other distributed systems. Estelle developed from efforts to specify protocols for Open Systems Interconnection (OSI). [DIAZ89, IS092] Estelle extends the syntax and semantics for the international standard for Pascal. The model underlying these Pascal extensions is a system of hierarchically-structured, communicating finite state machines (FSMs). Estelle FSMs, encapsulated as modules, may be active or passive. Active FSMs can run in parallel, communicating by exchanging messages. (Sharing of variables is supported between parent and child modules.)

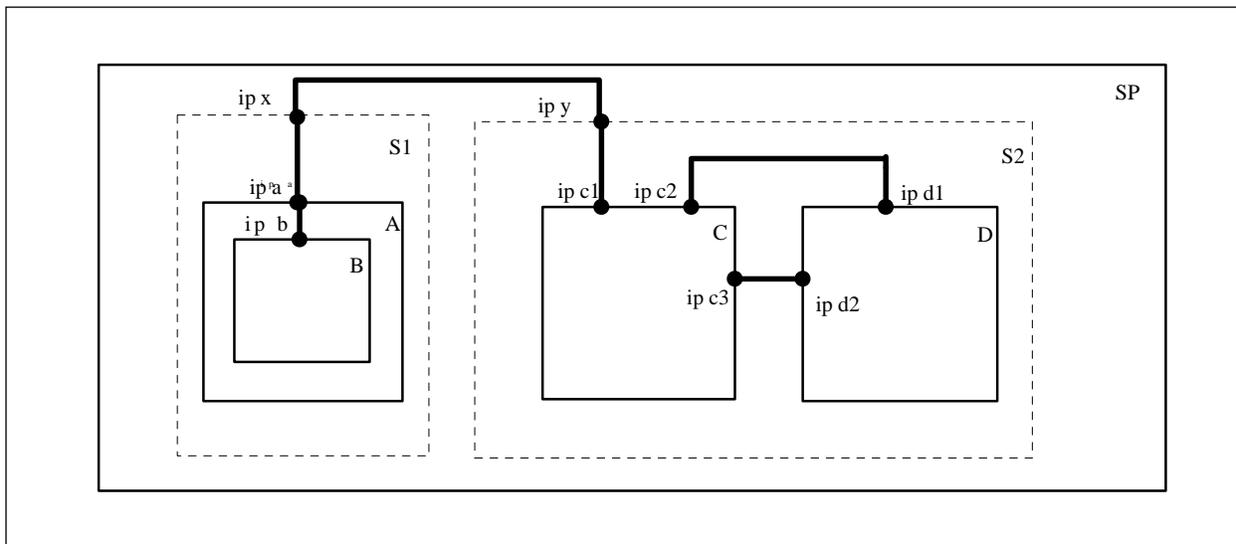


Figure VI-4. Example Estelle Specification Architecture [CHAM92, p. 6]

Interfaces between Estelle modules consist of three components. Interaction points, which can be external to peer modules or internal for parent-child modules, define the input and output points at which modules can communicate. Interaction points are unbounded, FIFO queues that can be point-to-point or shared (known as common queues in Estelle). Interactions comprise the messages that can be exchanged through an interaction point. All send operations in Estelle are non-blocking. Channels consist of two sets of interactions (in and out).

An Estelle specification comprises a hierarchy of module descriptions. Outer modules, each representing one physical node, can either of two types: `systemprocess` or `systemactivity`. Each `systemprocess` module can initiate subordinate active modules (process or activity) and each `systemactivity` module can initiate subordinate activity modules. A `systemprocess` permits multiple transitions to fire in subordinate modules during each firing cycle. A `systemactivity` allows only one enabled transition to fire during a firing cycle; when multiple transitions are enabled, one is selected non-deterministically for firing. Perhaps an example will help cut through the thicket of Estelle jargon.

Figure VI-4 shows an example of the architecture of an Estelle specification, SP. The specification consists of two systems, or nodes, S1 and S2. S1 and S2 each offer a single, external interaction point, ip x and ip y, respectively. These interaction points have been connected in the parent module, SP. System S1 consists of a single process, module A, that has another module, B, nested within it. Module B has an internal interaction point, ip b, that is attached to module A's interaction point, ip a, which is in turn attached to system S1's ip x. This connection graph implies that module B can exchange interactions with other nodes, in this case node S2.

System S2 consists of two processes, modules C and D. Only module C can exchange interactions with other systems. Modules C and D are connected through two pairs of interaction points (ip c2-ip d1 and ip c3-ip d2). Each participant at an interaction point must be assigned a named role that limits the allowable message types that the participant can send and receive. An Estelle channel is an interaction point with a named role and a designation of whether the queue is point-to-point or common.

The internal behavior of each active, Estelle module is specified using FSMs, extended with state-history variables and predicates that can guard transitions. Each transition represents an atomic set of actions. A delay construct is also included to represent the passage of time.

The syntax of Estelle modules, adapted from Pascal, includes a header and a body. For a given header, multiple bodies can be defined so that different implementations of the same interface can be instantiated. A module body consists of three parts: declarations, initializations, and transitions. Declarations comprise channels, nested modules, module variables, states and sets of states, and internal interaction points to children. The initialization portion defines the starting state, assigns variable values, starts any child modules, and connects and attaches interaction points. In Estelle, alternative initializations can be specified. The transition section describes the FSM that controls a module's behavior. The form of a transition is: **transition from** <state> **to** <state> [**when** <predicate> | **provided** <predicate>] [**delay** <time>]. Estelle modules can also use Pascal statements, along with some additional statements added for Estelle operations. Estelle-specific statements allow modules to be controlled (**init**, **release**, and **terminate**), enable interaction points to be managed (**connect**, **disconnect**, **attach**, **detach**), send interactions

```

specification EXAMPLE;

    default individual queue;
    timescale second;

channel UCH(User, Provider);
    by Provider: DATA_INDICATION;

channel NCH(User, Provider);
    by User: DATA_INDICATION;
    by Provider: SEND_ACK(x : integer);

module USER systemactivity;
    ip U: UCH(User);
end;

body USER_BODY for USER; external;

module RECEIVER systemactivity;
    ip U: UCN(Provider); N:NCH(Provider);
end;

body RECEIVER_BODY for RECEIVER; external; (*see Figure VI-6 *)

module NETWORK systemactivity;
    ip N: NCH(User);
end;

body NETWORK_BODY for NETWORK; external;

modvar X: USER; Y: RECEIVER; Z: NETWORK;

intialize
    begin
        init X with USER_BODY;
        init Y with RECEIVER_BODY;
        init Z with NETWORK_BODY;
        connect X.U to Y.U;
        connect Y.N to Z.N;
    end;

```

Figure VI-5. Example Estelle Specification -- Part I
 [ISO92, p. D.35]

(output), and add some convenient operators (**all**, **forone**, and

```

body RECEIVER_BODY for RECEIVER;
  (* declarations *)
  type time_period = integer;
  const high = 0;
         medium = 1;
         low = 2;
  state IDLE, AK_SENT;
  var   ak_no : 0..7;
         min, max, inactive_period : time_period;
  (* initializations *)
  intialize
    to IDLE
      begin
        min := 1; max := 20; inactive_period := 60;
        ack_no := 0;
      end;
  (* transitions *)
  trans
    from IDLE
      to IDLE
        priority medium
        when N.DATA_INDICATION
          name t1: begin
            output U.DATA_INDICATION;
            ak_no := ak_no + 1;
          end;
    to AK_SENT
      provied (ak_no > 0) and (ak_no <= 4)
      priority low
      delay(min, max)
        name t2: begin
          output N.SEND_AK(ak_no)
        end;
      provied (ak_no > 4) and ak_no < 7)
      priority high
      delay(min)
        name t3: begin
          output N.SEND_AK(ak_no)
        end;

  (* CONTINUED ON NEXT PAGE *)

```

Figure VI-6. Estelle Specification Of RECEIVER_BODY
[ISO92, p. D.40-D.41]

```

(* CONTINUED FROM PREVIOUS PAGE *)

    provided ak_no =7
priority high
    name t4:    begin
                output N.SEND_AK(ak_no)
            end;
provided otherwise;
priority low
    delay(inactive_period)
    name t5:    begin
                output N.SEND_AK(ak_no)
            end;

from AK_SENT
to IDLE
    name t6:    begin
                ak_no := 0;
            end;

end;

```

Figure VI-6. Estelle Specification Of RECEIVER_BODY - Cont.
[ISO92, p. D.40-D.41]

exist). An example will illustrate Estelle syntax.

Figure VI-5 gives part of an Estelle specification for a one-way receiver that acknowledges messages that are received. This specification, named EXAMPLE, defines two channels, UCH and NCH. Each channel allows two roles, a User and a Provider. On channel UCH, the Provider can send a DATA_INDICATION interaction, the User sends nothing (remember, the example is receive-only). On channel NCH, the User can send a DATA_INDICATION interaction, while the Provider can send a SEND_AK interaction containing a single integer. The specification also declares three nodes, USER, RECEIVER, and NETWORK. For each node, a channel is indicated and a role is defined. The RECEIVER is a Provider on channels UCH and NCH. The NETWORK is a User on channel NCH and the USER is a User on

channel UCH. Only the headers are given for each module; the bodies are specified elsewhere.

The specification, EXAMPLE, also declares three module variables, X, Y, and Z, of type USER, RECEIVER, and NETWORK, respectively. Then the three modules are instantiated with the indicated module bodies. Finally, the instantiated RECEIVER module is connected to the instantiated USER and NETWORK modules.

Figure VI-6 gives the Estelle specification of the RECEIVER_BODY declared in Figure VI-5. This shows how Estelle allows description of module behavior with an extended FSM. The module contains a type, some constants, and a few variables. Two states are declared: IDLE and AK_SENT. The initialization section should be self-explanatory, except that the time period for the delay for transition t5 begins ticking immediately because that transition is initially enabled (because transition t5 is enabled whenever the FSM is in the IDLE state).

The transition section is organized by state. When an DATA_INDICATION interaction arrives on channel N, transition t1 is triggered, causing a DATA_INDICATION to be sent on channel U and also incrementing ak_no. The transition does not change the explicit state of the FSM. Whenever the ak_no is greater than zero, additional transitions are enabled, but may be delayed. For example, transition t2 is enabled after the first DATA_INDICATION and stays enabled until the fifth DATA_INDICATION (when t2 is disabled) or until the delay interval expires (in which case t2 is fired). This FSM allows for up to eight messages to be received before an acknowledgment is sent, but will acknowledge a fewer number of messages when a certain period of time passes without eight messages being received. If no messages arrive in the inactive period, then an acknowledgment is sent (with an ack_no of zero) and the FSM spontaneously moves from the state AK_SENT to IDLE.

Estelle is supported by a number of tools, such as translators, run-time environments, graphical simulators, and syntax-directed editors. [DIAZ89] Estelle has been used to specify a range of communications protocols, including the international standard for distributed transaction processing. Although Estelle appears to offer reasonable means for specifying communications protocols, a number of enhancements have been proposed to increase Estelle's potential as a distributed systems design language.

FIFO queues are the only method currently allowed by Estelle for inter-module communication. Sijelmasi recommend adding a rendezvous to Estelle so that processes on the same node can interact via shared variables. [SIJE92] Sijelmasi also proposes named queues, exception handling, composition of input conditions, and more advanced queue operations. Chamberlain proposes four Estelle enhancements intended to extend its scope of application to general, distributed systems. [CHAM91, CHAM92] He cites the need for broadcast communications, n-way synchronization, a single-state history mechanism (for exception handling), and a strict, real-time constraint mechanism (to allow specification that an action must occur by a given time).

In summary, Estelle extends Pascal with constructs for defining hierarchically-structured, communicating FSMs. Estelle specifications are operational in nature; so a program implementing an Estelle specification is easy to build; and, in fact, several Estelle-to-C and Estelle-to-C++ translators exist. Enhancements to Estelle might make the language more suitable for specifying distributed systems.

E. Language Of Temporal Ordering Specification (LOTOS)

LOTOS merges two concepts (described in section V): temporal ordering and abstract data types. [ISO87, MUN91] Temporal ordering, based on a modification of the calculus of

```

type Bitstring is
  sorts Bit, BitString
  opns
    0, 1: -> Bit
    String : Bit -> BitString
    Append : BitString, Bit -> BitString
    Prefix : Bit, BitString -> BitString
    Concatenate : BitString, BitString -> BitString
  eqns
    forall x, y: Bit, s, t: BitString
      ofsort BitString
        Prefix(y, String(x)) = Append(String(y), x);
        Prefix(y, Append(s, x)) = Append(Prefix(y, s), x);
        Concatenate(t, String(x)) = Append(t, x);
        Concatenate(t, Append(s, x)) =
          Append(Concatenate(t, s), x);
endtype

```

Figure V-7. Example LOTOS Type Defined Using ACT-ONE
[MUNI, p.16]

communicating systems (CCS), is used to describe process behaviors and interactions. Abstract data types, as defined by the ACT-ONE language, are used to specify data structures and their operations. These two parts of LOTOS are independent and, in theory, a language other than ACT-ONE (Zed, for example), could be used to describe ADTs in LOTOS. Some researchers point out, however, that replacing ACT-ONE would require extensive modifications to the CCS portion of LOTOS. [LOGR88]

A LOTOS behavioral system consists of nested processes that interact via a multi-way rendezvous, where processes may withdraw before a rendezvous occurs. Message-passing protocols have been defined to allow the LOTOS rendezvous mechanism to be implemented between distributed processes. [SIST91] Because LOTOS formed the basis for explaining temporal ordering in section V, the discussion presented here will concentrate on the ADT portion of LOTOS and then on LOTOS tools and on the application of the language.

ACT-ONE provides an algebra for specifying LOTOS data types. [MUN91] Because ACT-ONE uses an algebraic notation to

describe ADTs, some terminology is different than described earlier for ADTs specified with FOQL. For example, LOTOS data types are called sorts and operations on data types are called relationships. All LOTOS operations are total functions. When a function has only two arguments, LOTOS permits either prefix or infix notation. In general, one builds a package containing several sorts, and uses these sorts as argument and result types in operations. Such a package of sorts and operations is called a type in LOTOS. A LOTOS type also includes equations that define invariant relations among the sorts and operations in the type. Perhaps this can best be explained using an example.

Figure V-7 gives a LOTOS description of a Bitstring type. BitString uses two sorts, Bit and BitString. Six operations are also defined. The operations 0 and 1 each return a Bit. The operation String casts an argument of Bit into a result of BitString. Prefix adds a Bit to the front of a BitString and Append adds a Bit to the end of a BitString. Concatenate joins two BitStrings to form a BitString. Four equations give the axioms of the operations. This definition should be familiar to the reader from the earlier description given of ADTs (refer to section V).

LOTOS types may be built from other types by using an import mechanism. Parameterized (i.e., generic) types can also be defined. LOTOS includes a pre-defined library of data types, for example, Boolean, NaturalNumber, NonEmptyString, Bit, and Octet. LOTOS ADTs suffer from several problems. For one, LOTOS does not support partial functions; thus, lacks arbitrary preconditions for operations and lacks the ability to constrain a sort to define subsorts. LOTOS also does not support the use of unspecified (i.e., generic) data algebras, so a specific algebra cannot be substituted into a larger specification.

Some tools have been implemented to support the LOTOS language. In section V, for example, a tool for translating

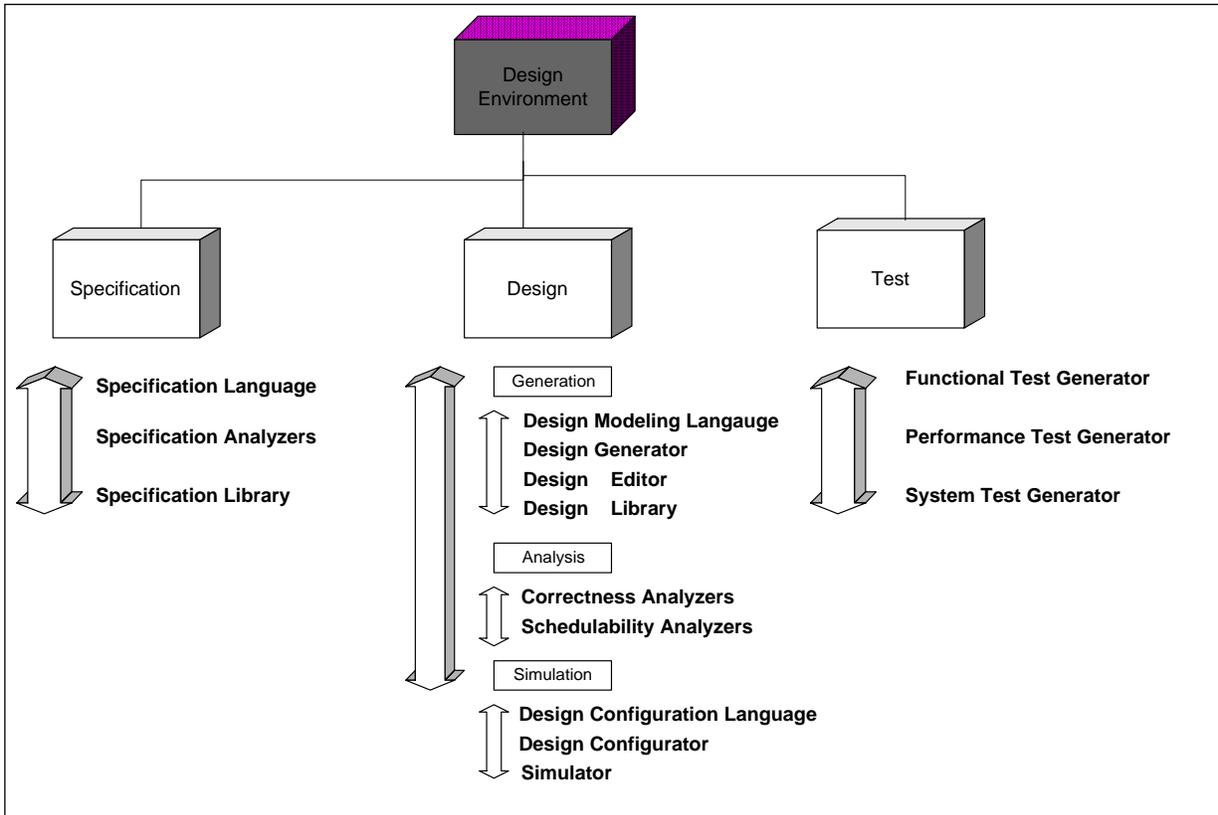


Figure VII-1. An Idealized Design Environment

LOTOS processes into C functions was described. A LOTOS interpreter, implemented in YACC/LEX, C, and Prolog, also exists. [LOGR88] The interpreter enables an analyst to: 1) recognize whether a given sequence of interactions is allowed by a specific LOTOS specification, 2) generate randomly chosen sequences of interactions from a LOTOS specification, 3) generate sequences from a specification under user guidance, and 4) simulate a specification, step-by-step. LOTOS specifications cannot be translated easily into efficiently executing programs. LOTOS allows infinite recursion, permits non-determinism, and enables the definition of unbounded axioms in ADTs. To use the interpreter at all, a specification must be carefully constructed. Interpreted LOTOS specifications cannot be used in lieu of a hand-coded implementation, yet LOTOS specifications

cannot be readily converted into hand-coded implementations because temporal ordering is used (see section V).

LOTOS has been applied. Several OSI protocol specifications are written in LOTOS. Other applications of LOTOS include specifications for computer-integrated manufacturing architecture [BIEM86] and communication security. [MUN91] More applications of LOTOS can be expected to occur.

VII. Design Environments

The preceding sections of this paper discussed the nature of software design, identified some key problems facing designers, and described a variety of formal models and methods proposed to help designers. In this section, the foregoing discussions are integrated. First, an idealized design environment is proposed. Then, design environments investigated by four research groups are described and evaluated.

A. An Idealized Design Environment For Real-Time Systems

An idealized design environment (IDE), illustrated in Figure VII-1, must support three activities: specification, design, and test. For specification, the IDE should help the designer to find, evaluate, and correct any omissions, ambiguities, and inconsistencies in the informal requirements. In addition, the IDE must offer help with the specification of timing constraints. Three tools are needed to help accomplish these objectives.

A specification language gives the designer a model for representing formally a system's functional and timing requirements. The language should be convenient to use, should be a medium for communication between the designer and the requirements analyst (and, if possible, between the analyst and the user), and should be based on a formal model that is amenable to analysis. Although multiple forms of analysis might

be required, a single specification language should suffice if at all possible. Multiple translations of the specification should be avoided, even when provided with automated translators, because errors can creep in among the various representations.

The IDE should offer specification analyzers that can identify and evaluate requirements errors. The exact nature of such analyzers is an open area for research.

A specification library should be included in the IDE to support reuse of formal specifications. Such a library might also be used to record historical data on specification errors as they are encountered and resolved.

For design, three categories of tools are needed. Generation tools can help a designer produce a concurrent design from a formal requirements specification; analysis tools enable a designer to assess the correctness of a proposed design; simulation tools allow a designer to exercise a proposed design under varying loads and in alternative configurations. Each category of tools is considered further below.

The key to design generation is a design modeling language (DML) and the underlying model and representation that support the language.

[T]he most important component [in making real-time systems easier to understand] is the development of the underlying models used to represent the systems. Such a model should ideally possess formal semantics that allow a system's correctness to be verified. At the same time, it should represent the software and real-world entities in a way that feels natural to system designers. [BIHA92, p. 26]

The DML and underlying model should be consistent with the design method used. This will enable the designer to think and act in familiar terms. The representation underlying the language and model must support efficient analysis and

simulation without requiring additional translation (unless such translation is automatic).

A design generator should provide automated assistance to convert a formal specification into a concurrent, and possibly distributed, design. A design generator should apply the rules of a design method when generating the design. The generator should consult with the designer for guidance as the design is created. The resultant design should show the structure of the software, as distribution units, tasks, and information hiding modules, and should generate a skeleton for task behaviors and module definitions. A design editor should enable the designer to modify the design and to add the details needed to complete the generated skeletons. An editor should also allow a designer to enter a design without assistance from the generator.

A design library can support the generation of task and module skeletons. A library, coupled with a search program and the design editor, might also support the reuse of previous designs.

Once a design exists, analysis is needed. If possible, automated correctness analyzers should be employed. Approaches to design verification are the subject of much current research. Some approaches rely on exercising the design; others attempt to mathematically evaluate the design. Exercising a design, much as testing, can detect errors, but cannot guarantee the absence of errors. On the other hand, analyzing complex designs for real-time systems can be mathematically difficult, labor intensive, error prone or computationally infeasible. For analysis, a design must usually be translated into a formal model that reduces complexity. The translation process can alter the design so that the resulting analysis might not apply to the actual system design.

Schedulability analyzers using rate monotonic analysis can be applied to concurrent designs. The designer must, however,

supply accurate information to generate reliable results. Having a schedulability analyzer available can allow the design to be assessed iteratively as more detailed information becomes available. Schedulability analyzers should also enable the designer to specify event threads for which response time estimations are needed.

While analysis can help the designer evaluate the correctness and schedulability of proposed designs, simulation allows a design to be exercised under load, and in a range of configurations. To support simulation, the IDE should contain a design configuration language (DCL), a design configurator, and a simulator. The DCL lets the designer allocate distribution units to nodes, specify the performance characteristics of the nodes and of the communications links between them, and to constrain any system resources. The design configurator analyzes DCL files and the design model and then generates a design simulation model. The simulator exercises interactively the configured model. Interactive execution enables a designer to investigate the run-time behavior of a proposed design. The simulator should detect correctness violations, record performance characteristics, and monitor resource usage. The designer should be given complete freedom to alter a configuration while the simulation is running, as well as to halt the simulation and examine the state of the system.

All analysis and simulation should operate from the design, as specified by the DML and underlying model. A designer should not be required to produce different models of the design for each purpose. Researchers are currently investigating approaches to accomplish these objectives.

The final activity that an IDE should support is testing. An IDE should allow automation-assisted test generation from a formal requirements specification. Three types of tests should be generated: functional, performance, and system. Proper test

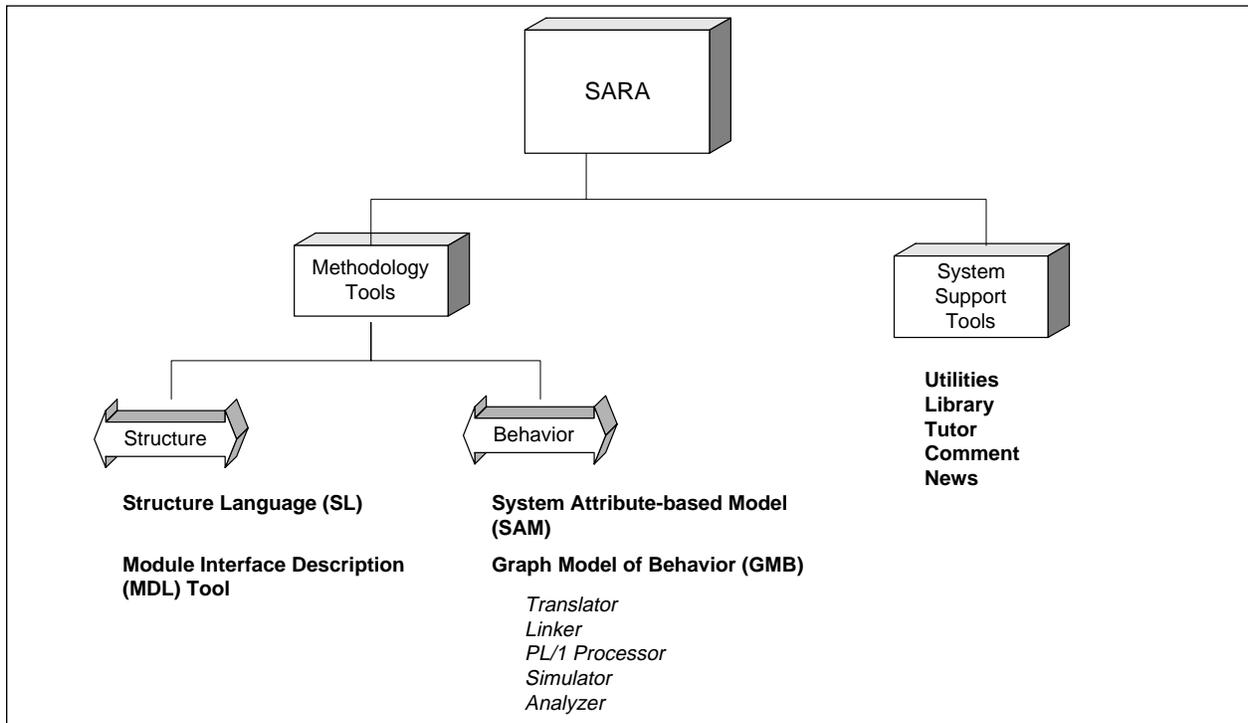


Figure VII-2. SARA Design Environment [ESTR86, p. 294]

generation can enable an intelligent and cost effective testing campaign. The generated tests can be used by software test teams to support integration and system testing. In addition, since the tests can be made available during software design, the tests can guide the designers analysis and simulation of design alternatives. Methods for generating appropriate software tests are currently the subject of much research.

Beyond the specifics covered so far, a few desirable qualities for an IDE can be mentioned. The languages available in an IDE (i.e., the specification language, DML, and DCL) should be familiar and comfortable to the designer. If possible, these languages should support existing design methods, rather than force the designer to learn complex, new approaches. A balance is required between two extremes. On one hand, since current design methods appear to lack rigor and formal semantics, the ability of an IDE to generate, analyze,

and simulate designs might prove too limited. On the other hand, introducing a high degree of rigor and formality will likely require new design methods. Such new methods might prove time-consuming to learn, difficult to apply, and limited in scope. A successful approach might start from existing design methods and add, unobtrusively, the rigor and formality needed to support design generation, analysis, and simulation.

The tools in an IDE should be easy for a designer to use. If these tools are not easy to use, then they will not be used. (Of course, ease of use can be traded to some degree for improvements in effectiveness.) IDE tools should also be efficient. IDEs should provide, almost as a side effect, traceability from informal requirements, to formal requirements, to tests, and to the design.

The idealized IDE, then, must achieve some difficult objectives. The reader probably understands that no such IDE exists. Researchers have, however, investigated design environments for a number of years. Below, four proposed design environments are described and evaluated.

B. System ARchitects Apprentice (SARA)

The System ARchitects Apprentice, SARA, a joint development of researchers at UCLA and the University of Wisconsin, provides an interactive environment for modeling, analyzing, and simulating designs for concurrent systems. [ESTR86] SARA's goals are six: 1) to allow reasoned consideration of hardware and software tradeoffs, 2) to support building models of a system's operating environment, 3) to separate structure from behavior, 4) to enable early detection of design flaws, 5) to facilitate composition, implementation, and testing of designs, and 6) to assist individual designers in a manner most comfortable to them.

The SARA environment comprises two categories of tools, as illustrated in Figure VII-2. The main SARA tools allow designers to model structure and, separately, behavior. Using the Structure Language (SL) designers can specify a fully nested, hierarchical structure of modules and module interconnections (via sockets). The Module Interface Description (MID) tool provides access, via program code, to named, design resources. Ada, for example, can be a satisfactory MID language.

Using the behavior tools designers can specify, analyze, and simulate a design. The underlying behavioral model is based on the UCLA Graph Model of behavior (GMB).⁵ SARA uses a formal GMB to model control and data flows, and the interpretation of data types. To model control, designs are specified with nodes and control arcs in a manner similar to Petri nets. The data domain is modeled using processors (i.e., transforms), data sets (i.e., data stores), and data arcs (i.e., data flows). SARA's data domain model represents data flow diagrams. In the interpretation domain, SARA can model the data types of data sets and of algorithms in node-processor pairs. In essence, the interpretation model is similar to a data dictionary and associated mini-specifications.

Once a behavioral and structural model are specified for a design, SARA allows the designer to investigate a range of issues. For example, the behavioral model can be merged with an environmental model, after which simulation experiments can be conducted. The SARA simulator is interactive and includes a range of nice features, plus built-in checking for specific design flaws. Or, the control and data graphs can be analyzed to detect contention for resources. The designer might also blend in the interpretation model to attempt validation of the

⁵ The UCLA Graph Model of behavior was not covered in section V. Suffice to say that the GMB, with appropriate restrictions, is equivalent to a Petri net model. [ESTR86, p.294]

entire design. Consider further some of the dynamic modeling capabilities of SARA.

The GMB model enables the designer to represent explicitly contention for active and passive resources. Once encoded in a design, the contention is modeled in SARA analyses for correctness and for performance. To assess correctness, a control flow analyzer builds a reachability graph. Since these graphs can grow quite large (or even be infinite), a strong reduction algorithm is used to reduce the state space without sacrificing too much analysis. The reduction algorithm compacts sequential paths and paths guaranteed never to deadlock.

To assess performance, the GMB simulator derives stochastic queuing models from the design specification. Using the queuing model, SARA can estimate, for each modeled resource, the following average values (within a known confidence interval): utilization, queue size, and waiting time. SARA can also determine distributions for queue size and waiting time.

The diverse tool set provided by SARA is presented through a coherent, single user interface. Unfortunately, the syntax of the various languages appear difficult to master. Another shortcoming of SARA is the large, cumbersome, and complex nature of the software. A designer must carefully consider the goals of a particular study and build models appropriately because SARA supports a range of evaluation methods. SARA could benefit from inclusion of an expert system to guide designers through the SARA design process. A graphical interface could also make SARA more approachable. As with many tools described in the literature, SARA needs improved means to model time-constrained systems and to avoid the combinatoric explosion problem faced when analyzing realistic designs.

C. Methodology for Integrated Design And Simulation (MIDAS)

Bagrodia and Shen describe a design methodology for integrated design and simulation (MIDAS) that differs sharply

from SARA. MIDAS "...supports the design of distributed systems via iterative refinement of hybrid models". [BAGR91, p. 1042] A hybrid model is a partially implemented design; some components comprise simulation models, others consist of operational code. The main purpose of MIDAS is to assess the average performance traits of a design. MIDAS increases modeling realism by representing interrupts and distributed design components.

MIDAS development begins with construction of a discrete-event simulation model that the designer transforms over time to an operational system. Design components are modeled using a concept called partially implemented performance specifications (PIPS). PIPS are implemented using an existing simulation language, such as MAY or Maisie, but with extensions to allow interface to components coded in a high-level programming language. Much attention is given to interleaving simulated and live execution so that interrupts can be properly modeled, especially since MIDAS supports distributed execution of models.

Some shortcomings to MIDAS are readily apparent. No support is provided for analysis of correctness. The performance modeling available with MIDAS can only predict average behavior. In fact, because MIDAS allows hybrid modeling, the simulation hardware must be identical to, or scalable to, the hardware on which the real system will execute, if an accurate performance prediction is needed. MIDAS requires extension to allow modeling of hard, real-time systems.

D. PROTOB

PROTOB aims to facilitate executable specifications for large-scale, event-driven systems. [BALD91] The intent of PROTOB is twofold: 1) to enable behavioral prototyping and performance evaluation of a software specification and 2) to support automated translation of a specification into an architecture design. To accomplish these objectives, PROTOB

provides tools for modeling, application generation, and emulation.

PROTOB includes two specification languages, one graphic, one textual, that describe formally object behavior. The specification languages are based on a form of high-level Petri nets, called PROTnets, integrated with extended data flows and high-level programming languages. A PROTOB specification embodies an executable, object model, where each object encapsulates a PROTnet. PROTOB objects can be constructed, hierarchically, from other PROTOB objects, and can communicate with each other via message passing.

Application generation tools, part of a complete CASE environment supporting PROTOB, translate PROTOB specifications into an executable program, either centralized or distributed, implemented in C or Ada. PROTOB specifications can be translated into a simulator or an emulator. The simulator version can exercise the system while including the modeling of time. The emulator version is a prototype that must be integrated with an actual environment. Simulations are used to evaluate the performance of a specification, while emulations are used to evaluate control behavior. Both types of dynamic model can be generated from the same PROTOB specification, and both are executed by an underlying inference engine, tailored especially for PROTnets. To see how this might be achieved, consider the details of a PROTOB object specification.

Each PROTOB object is defined with a script; a script is simply a text file with definitions of token types, local variables, and actions associated with each PROTnet transition. The variables and actions are written in C or Ada, depending on the target language of the application. Each script, in detail contains:

- ♦ token types, structured messages (like Ada records);
- ♦ communication types, used to connect objects;

- ♦ object parameters, which must be scalar values;
- ♦ local variables, which can be initialized;
- ♦ declaration of external functions (in the target language);
- ♦ transition definitions, with optional predicates and actions (in the target language) and optional priority;
- ♦ optional initialization actions; and
- ♦ optional final actions.

PROTnet transitions are responsible for introducing timing constraints into the model. Each transition can include, optionally, delayed release and delayed firing parameters.

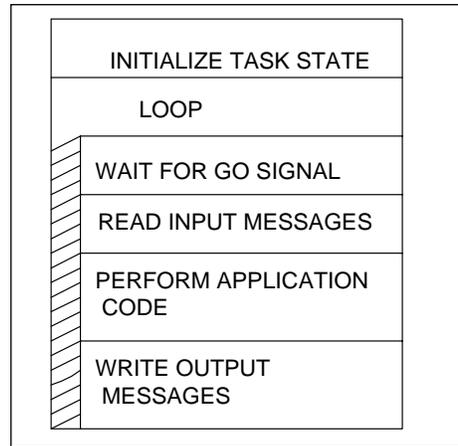


Figure VII-3. A Mars Task Wrapper

PROTOB appears to provide a useful design environment for distributed, real-time systems. A number of successful applications of PROTOB are reported, e.g., manufacturing systems, monitoring systems, and communications protocol design, with complexity ranging from 10 to 70 different objects with up to 50 transitions each. [BALD91, pp. 829-830]

The emphasis of PROTOB is on specifying and exercising a system, not on analysis. Thus, as with other such approaches, errors can be detected, but the absence of errors cannot be shown.

E. Mars

Mars, a product of researchers at the Technical University of Vienna, encompasses a prototype development support environment for maintainable, real-time systems. [POSP92] The aim of Mars is to facilitate construction of hard-real-time systems that are understandable and maintainable, as well as correct and timely. Mars supports the approach of pre-run-time scheduling. An off-line scheduler produces, if feasible, a task schedule and allocates messages to bus slots (Mars supports

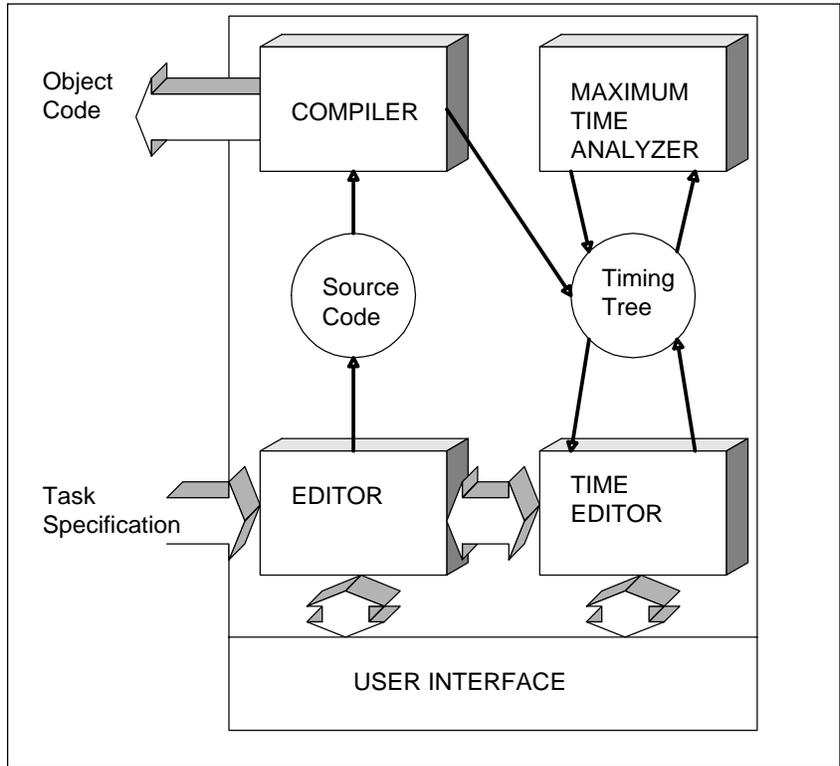


Figure VII-4. Mars Programming Environment [POPS92, p.38]

design of multiprocessor systems connected by a high-speed bus) in a manner that meets all synchronization, timing, and communication constraints. In a very controlling fashion, the Vienna researchers attempt to eliminate every source of uncertainty so that Mars designs are

deterministic and completely under a designer's control. They even, for example, avoid the problems posed by instruction caching and pipelining in modern, high-performance microprocessors. They avoid this by refusing to use them - every target processor in their system is a fixed instruction Motorola 68000. Each board in the network consists of two processors: one for applications and one for low-level direct-memory access in support of bus communications processing. Every operation in the network happens synchronously. The only system interrupt is a clock tick to mark the boundary of CPU and bus slots. Although a message passing paradigm is used for inter-processor communications, each message gets a dedicated time slot on the bus and each task gets a cyclic execution slot on its local processor. All of this is predetermined by the off-line scheduler. This

background is important to understand before the software design approach and environment are discussed.

Every task in a Mars design is specified in a real-time language, Modula R, derived from Modula 2. The skeleton of each task consists of a wrapper logic that is identical for every task in a Mars design. The task skeleton is shown in Figure VII-3. Each task loops forever performing four actions: wait for the start signal (i.e., the task's schedule slot to arrive), read all input messages, perform application specific operations, and write any output messages. The application specific operations are describe in Modula R. Modula R restricts Modula 2 by eliminating GO TO, abolishing recursion, and banning dynamic memory allocation from the heap. All of these operations would introduce uncertainty into a task's timing characteristics. Modula R also adds some primitives to Modula 2. The additions, **scopes**, **markers**, and **loop sequences**, let a programmer express knowledge about infeasible paths. Adding this meta-knowledge, unknown to all but the programmer, can tell Mars that a program will leave a control structure within a finite bounds. The Mars compiler system can analyze program code (when augmented appropriately with **markers**, **loop sequences** and **scopes**) to determine a task's timing behavior.

After a program is analyzed, the source code is displayed in an editor with each control construct annotated with an estimated program execution time. The total execution time for the task is computed from the sum of the calculated time, coupled with the system's understanding of time for repetitive, overhead operations (e.g., context switching and message passing). Through a novel concept, called time editing, the programmer can suggest different hypothetical maximum times where she thinks that the time can be improved through code tuning or by choosing, ultimately, a more efficient algorithm.

Once the programmer has overridden the analyzer's estimates, the programmer's figures will be used in all further calculations.

A general layout of the Mars development environment is shown in Figure VII-4. Mars supports either top-down or bottom-up design as appropriate to the problem and to the designer's wishes. Mars allows designers to incorporate execution timing considerations into the design from the start. (Of course, these are timing budgets assigned to program elements, not time constraints entered from a requirements specification.) After a timing data base exists for the tasks in the design, an off-line scheduler can allocate tasks to processors, can arrange synchronized access to resources, and can budget bus bandwidth for inter-node message communications.

As the reader can see, a variety of approaches exist to supporting real-time system designers with automation. Each approach as described above has strengths and weaknesses, and each fails to achieve all the characteristics of the IDE detailed at the beginning of the section. Although some of these environments offer useful aids, none has achieved successful application in the design of large, commercial, real-time systems.

VIII. Conclusions

This paper described the purpose of design as threefold: 1) to discover the structure of a problem, 2) to create outlines, or architectures, of a solution, and 3) to evaluate the solutions against the problem. For designers of software systems, these goals can be translated into some specific steps. First, informal software requirements specifications must be reviewed and analyzed using some systematic method. Second, the set of software components, and relationships between them, necessary to meet the requirements must be described in enough

detail to permit accurate evaluation and implementation. The third purpose of design, evaluation, is often handled poorly for software systems. Typically, software designs are evaluated only during system testing.

For designers of real-time, software systems, functional requirements are augmented by timing constraints. Such real-time constraints must be satisfied for a software system to be considered correct. The timing constraints generally fall into two classes: response-time for specific events and system throughput in the face of a peak load.

When a real-time, software system is also distributed several additional concerns arise. Processes and data must be allocated among nodes in the system. An inter-node message passing paradigm must be defined. A means must be devised to integrate the inter-node and intra-node message passing models. The physical characteristics of inter-node communications paths must be accounted for. Incompatible data representations must be reconciled. System security issues must be identified and resolved.

The problems recounted above reveal a number of challenges for researchers who seek to improve the lot of software designers, particularly designers of distributed, real-time software. Designers need methods to detect and resolve flaws contained in software requirements specifications. Designers and specifiers need improved mechanisms for specifying software system timing requirements. Designers could benefit from approaches to bound the maximum communications delay and residual error rate between nodes in distributed, real-time systems. Designers would profit from an accepted paradigm, with well-defined semantics, for inter-task communication among distributed, real-time nodes. Designers could produce more effective designs if assisted by methods to enable dynamic evaluation of alternative designs.

The current state-of-practice in real-time software design appears to rely on methods developed two or three decades ago. A real-time designer partitions software into modules and then, perhaps with the aid of a pre-run-time scheduler, schedules the modules to execute in a specific, cyclic order that meets all synchronization and timing constraints. The resulting software is often difficult to understand, to maintain, and to expand.

Newer, concurrent design approaches emphasize understandability, maintainability, and expandability at some cost in deterministic performance. Recent developments concerning rate monotonic scheduling theory promise to enable concurrent designs to achieve effective real-time performance. Currently, however, effective use of rate monotonic analysis requires support from underlying operating system mechanisms that are not yet implemented widely.

This paper found that most design approaches used by practitioners lack a formal semantic model. Without such a model, analyzing and evaluating alternative designs will remain difficult. For this reason, much of the current research surrounding software specification, design, and evaluation investigates the application of formal models and methods. In general, formal models for design can be viewed as behavioral models or structural models. Behavioral models include finite state automata, Petri nets, temporal ordering, and executable specifications. These models emphasize the control aspects of a design. Behavioral models usually possess several shortcomings: 1) they do not, generally, include the notion of time, 2) they are subject to state explosions which can make analysis computationally infeasible, 3) when they are augmented with higher-level constructs to improve notational convenience, they lose some of their analytical properties, 4) they often incorporate variations to support specific needs, and 5) they sometimes require the designer to learn a difficult syntax.

Structural models include abstract data types, axiomatic methods, and temporal logic. These models emphasize the static properties of a design and can provide a basis for proving that an implementation exhibits the desired properties. Structural models share a number of weaknesses. Writing formal specifications is a difficult, labor intensive activity. Some structural models cannot be used to describe the behavioral or correctness properties of sequential tasks. When sequential tasks can be described, most formal models do not account for specification of timing constraints.

Application of formal models, behavioral or structural, requires a syntactic model amenable to use by designers. Several research efforts aim to build a suitable syntax on the foundation of formal models. Communicating Sequential Processes (CSP) led to several languages that, while never gaining a foothold with practicing designers, influenced international standard specification languages such as Estelle and LOTOS. Estelle extends Pascal with a formal model of communicating finite state automata. The result is an understandable specification language that perhaps demands the inclusion of too much implementation detail. LOTOS merges a temporal ordering model with an abstract data type (ADT) model to produce a language capable of specifying both behavior and structure. Unfortunately, LOTOS, while a powerful specification language, embodies several disadvantages. For example, the LOTOS ADT language does not support partial functions nor arbitrary preconditions for operations. Also, LOTOS specifications cannot be translated easily into efficient implementations.

A number of ambitious research projects attempt to combine a range of tools to compose software design environments. This paper presented an idealized design environment (IDE) and then examined four proposed design environments against the IDE. The IDE contained tools for specification (including a language,

analyzers, and a library), for generating, analyzing, and simulating designs, and for generating tests (functional, performance, and system). Design generation tools include a design modeling language, a design generator, a design editor, and a design library. Design analyzers were included for evaluating correctness and schedulability. Design simulation tools included a design configuration language, a design configurator, and a simulator.

Among the design environments discussed in this paper, SARA and PROTOB contained the richest set of tools. SARA exhibits several shortcomings. The syntax of various SARA languages appears difficult to master; the tools set is large and cumbersome; the user interface is not very friendly; time constraints are not modeled; SARA behavioral models tend to suffer state explosion; the designer must build separate SARA models to evaluate different aspects of a design.

PROTOB overcomes many of the shortcomings of SARA, but sacrifices some capabilities. Mainly, PROTOB cannot be used to analyze a design for correctness; a design can be exercised, however, and errors can be detected. In addition, PROTOB designs can be simulated and can form the basis for generating implementations in a high-level language such as C or Ada.

The design environment presented by Mars attempts to codify a deterministic design within a synchronous hardware/software system. Mars designs are to be built from identical, simple hardware components (i.e., high-speed buses and Motorola 68000 processors) and software components (i.e., Mars tasks). The designer's main job is to specify application specific logic wrapped with Mars tasks and to hone the timing of that logic to meet the performance objectives of the application. To successfully apply Mars a designer must work within the frame provided; thus, Mars designs cannot generally be moved to other hardware and software environments.

Of the four design environments studied, MIDAS was most limited. The main goal of MIDAS is to enable simulation of a design and then to allow that simulation to be progressively elaborated into an implementation. MIDAS enables prediction of average performance, but not of worst-case performance. MIDAS provides no support for analysis of correctness. As with Mars, MIDAS requires that the simulation hardware be identical to the hardware on which the real system will execute.

These representative design environments, SARA, PROTOB, Mars, and MIDAS, collectively illustrate the immaturity of the current state of research regarding software design; however, some avenues appear promising. A design environment should begin with a modern design method that is known to practitioners. Tools added to the design environment should work from a syntax familiar to designers. A design environment must include some means for identifying and repairing the ambiguities, omissions, and inconsistencies present in informal requirements documents. In most cases, the means for achieving these ends requires translation of informal requirements into a formal, requirements model. Tools should be included in a design environment to help a designer create a design from a formal, requirements model.

For most complex designs, tools to analyze functional correctness appear computationally infeasible; thus, an emphasis should be placed on design exercisers and simulators. Such exercisers will require an underlying semantic model of essential design details. A method to analyze the schedulability of a design must also be included in any design environment.

Methods exist to generate concurrent designs for distributed, real-time systems. The use of such methods is inhibited by an inability to predict the worst-case performance of the resulting designs. (Rate monotonic scheduling theory

shows potential to overcome this barrier.) In addition, alternative designs are often not considered because no means exists to evaluate one design against another. (Such evaluation is particularly difficult for concurrent designs.) Usually, designers must await system testing to discover design flaws that might require major redesign and re-implementation of software. Finding and correcting design problems as early as possible should improve the quality and reduce the cost of distributed, real-time software.

IX. References

A. General Concepts of Real-Time and Distributed Systems

- [AMBL92] A. L. Ambler, et al., "Operational Versus Definitional: A Perspective on Programming Paradigms", *IEEE Computer*, September 1992, pp. 28-43.
- [BIHA92] T. E. Bihari and P. Gopinath, "Object-Oriented Real-Time Systems: Concepts and Examples", *IEEE Computer*, December 1992, pp. 25-32.
- [GOMA86] H. Goma, "Software Development Of Real-Time Systems", *Communications of the ACM*, July 1986, pp. 657-668.
- [KLEI85] L. Kleinrock, "Distributed Systems", *IEEE Computer*, November 1985, pp. 90-103.
- [NATA92] S. Natarajan and W. Zhao, "Issues in Building Dynamic Real-Time Systems", *IEEE Software*, September 1992, pp. 16- 21.
- [SHAT84] S. M. Shatz, "Communication Mechanisms for Programming Distributed Systems", *IEEE Computer*, June 1984, pp. 21-28.
- [STAN82] J. A. Stankovic, "Software Communication Mechanisms: Procedure Calls Versus Messages", *IEEE Computer*, April 1982, pp. 19-25.
- [STAN88] J. A. Stankovic, "Misconception About Real-Time Computing - A Serious Problem for Next-Generation Systems", *IEEE Computer*, October 1988, pp. 10-19.

[SUMM89] R. C. Summers, "Local-area Distributed Systems", *IBM Systems Journal*, Vol. 28, No. 2, 1989, pp. 227-240.

B. Scheduling and Performance of Real-Time and Distributed Systems

[CHU91] W. W. Chu, et al., "Task Response Time For Real-Time Distributed Systems With Resource Contentions", *IEEE Transactions on Software Engineering*, October 1991, pp. 1076-1092.

[EIND87] P. Ein-Dor and J. Feldmesser, "Attributes Of The Performance Of Central Processing Units: A Relative Performance Prediction Model", *Communications of the ACM*, April 1987, pp. 308-317.

[FAUL88] S. R. Faulk and D. L. Parnas, "On Synchronization In Hard-Real-Time Systems", *Communications of the ACM*, March 1988, pp. 274-287.

[GAFF91] J. D. Gafford, "Rate Monotonic Scheduling", *IEEE Micro*, June 1991, pp. 34-38.

[JOES86] M. Joseph and P. Pandya, "Finding Response Times in a Real-Time System", *The Computer Journal*, Vol. 29, No. 5, 1986, pp. 390-395.

[KENN91] K. B. Kenny and K. Lin, "Measuring and Analyzing Real-Time Performance", *IEEE Software*, September 1991, pp. 41-49.

[LIEN92] C. Lien and C. Yang, "Specification and Quality Assurance of Timing Constraints in Real-Time Systems Development", *Software--Practice and Experience*, November 1992, pp. 963-984.

[LIU90] L. Y. Liu and R. K. Shyamasundar, "Static Analysis of Real-Time Distributed Systems", *IEEE Transactions on Software Engineering*, April 1990, pp. 373-388.

[MAHJ84] A. Mahjoub, "On the Static Evaluation of Distributed Systems Performance", *The Computer Journal*, Vol. 27, No. 3, 1984, pp. 201-208.

[OBEN93] R. Obenza, "Rate monotonic analysis fro real-time systems", *IEEE Computer*, March 1993, pp.73-74.

- [PENG93] D. Peng and K. Shin, "Optimal Scheduling of Cooperative Tasks in a Distributed System Using an Enumerative Method", *IEEE Transactions on Software Engineering*, March 1993, pp. 253-267.
- [POSP92] G. Pospischil, et al., "Developing Real-Time Tasks with Predictable Timing", *IEEE Software*, September 1992, pp. 35-44.
- [SHA90] L. Sha and J. B. Goodenough, "Real-Time Scheduling Theory and Ada", *IEEE Computer*, April 1990, pp. 53-62.
- [SHEP91] T. Shepard and J.A. Martin Gagne, "A Pre-Run-Time Scheduling Algorithm For Hard Real-Time Systems", *IEEE Transactions on Software Engineering*, July 1991, pp. 669-677.
- [SEI92] Software Engineering Institute, The Handbook of Real-Time Systems Analysis: Based on the Principles of Rate Monotonic Analysis, Technical Report (Draft), July 1992, approximately 200 pages.
- [STAN91] J. A. Stankovic and K. Ramamritham, "The Spring Kernel: A New Paradigm For Real-Time Systems", *IEEE Software*, May 1991, pp. 62-72.
- [Xu93] J. Xu and D. L. Parnas, "On Satisfying Timing Constraints in Hard-Real-Time Systems", *IEEE Transactions on Software Engineering*, January 1993, pp. 70-84.

C. Design Methods For Real-Time Systems

- [BECK89] S. A. Becker and A. R. Hevner, "Concurrent System Design With Box Structures", Proceedings of the 11th International Conference on Software Engineering, 1989, pp. 32-40.
- [BENV91] A. Benveniste and G. Berry, "The Synchronous Approach to Reactive and Real-Time Systems", *Proceedings of the IEEE*, September 1991, pp. 1270-1282.
- [FRAN90] G. Frank and J. DiSanto, "Software/Hardware codesign of real-time systems with ADAS", *Electronic Engineering*, March 1990, pp. 95-102.

- [FREE80] P. Freeman, "The Nature Of Design", Tutorial on Software Design Techniques, (Freeman and Wasserman, editors), IEEE Computer Society, April 1980, pp. 46-53.
- [GOMA84] H. Gomaa, "A Software Design Method For Real-Time Systems", *Communications of the ACM*, September 1984, pp. 938-949.
- [GOMA89] H. Gomaa, "A Software Design Method For Distributed Real-Time Systems", *Journal Of Systems And Software*, February 1989.
- [HULL91] M. Hull, et al., "Development Methods for Real-Time Systems", *The Computer Journal*, Vol. 34, No. 2, pp. 164-172.
- [KURK93] R. Kurki-Suonio, "Stepwise Design of Real-Time Systems", *IEEE Transactions on Software Engineering*, January 1993, pp. 56-69.
- [LEVI90] S-T Levi and A. K. Agrawala, Real-Time System Design, McGraw-Hill, New York, 1990, 299 pages.
- [NIEL87] K. W. Nielsen and K. Shumate, "Designing Large Real-Time Systems With Ada", *Communications of the ACM*, August 1987, pp. 695-715.
- [NIEL90] K. W. Nielsen, Ada in Distributed Real-Time Systems, McGraw-Hill, New York, 1990, 371 pages.
- [RAVN93] A. P. Ravn, et al., "Specifying and Verifying Requirements of Real-Time Systems", *IEEE Transactions on Software Engineering*, January 1993, pp. 41-55.
- [RIDDD80] W. E. Riddle, "An Event-based Design Methodology Supported By Dream", Tutorial on Software Design Techniques, (Freeman and Wasserman, editors), IEEE Computer Society, April 1980, pp. 269-283.
- [ROFR92] J. J. Rofrano, Jr., "Design considerations for distributed applications", *IBM Systems Journal*, Vol. 31, No. 3, 1992, pp. 564-589.
- [SAND89a] B. I. Sanden, "An Entity-Life Modeling Approach To Design Of Concurrent Software", *Communications of the ACM*, March 1989, pp. 330-343.
- [SAND89b] B. I. Sanden, "Entity-Life and Structured Analysis in Real-Time Software Design -- A Comparison",

Communications of the ACM, December 1989, pp. 1458-1466.

- [SAND93] B. I. Sanden, "Designing Control Systems With Entity Life Modeling", unpublished manuscript, June 23, 1993, 21 pages.
- [SIMO81] H. A. Simon, The Sciences of the Artificial, The MIT Press, Cambridge, Mass., 1981, 247 pages.
- [WITT85] B. I. Witt, "Communication Modules: A Software Design Model for Concurrent Distributed Systems", *IEEE Computer*, January 1985, pp. 67-77.
- [YAMA93] S. Yamazaki, et al., "Object-Oriented Design of Telecommunications Software", *IEEE Software*, January 1993, pp. 81-87.

D. Formal Methods

- [HOAR87] C. A. R. Hoare, "An Overview of Some Formal Methods for Program Design", *IEEE Computer*, September 1987, pp. 85-91.
- [IPSE90] E. A. Ipser and D. S. Wile, "A Multi-Formalism Specification Environment", *Software Engineering Notes*, December 1990, pp. 94-106.
- [WING90] J. M. Wing, "A Specifier's Introduction to Formal Methods", *IEEE Computer*, September 1990, pp. 8-24.

D.1 Finite State Machines

- [CHAM91] S. Chamberlain and P. Amer, "Broadcast Channels in Estelle", *IEEE Transactions on Computers*, April 1991, pp. 423-436.
- [CHAM92] S. Chamberlain, Estelle Enhancements for Formally Specifying Distributed Systems, University of Delaware, Dissertation, TR 92-17, December 1992, 170 pages.
- [CHAN85] M. Chandrasekharan, et al., "Requirements-Based Testing of Real-Time Systems: Modeling for Testability", *IEEE Computer*, April 1985, pp. 71-80.
- [COLE92] D. Coleman, et al., "Introducing Objectcharts or How to Use Statecharts in Object-Oriented Design", *IEEE*

Transactions on Software Engineering, January, 1992, pp. 9-18.

- [DIAZ89] M. Diaz, et al. (editors), The Formal Description Technique Estelle, North-Holland, 1989, 439 pages.
- [HARE87] D. Harel, et al., "On the Formal Semantics of Statecharts", *IEEE*, 1987, pp. 54-60.
- [HARE90] D. Harel, et al., STATEMATE: A Working Environment for the Development of Complex Reactive Systems", *IEEE Transactions on Software Engineering*, April 1990, pp. 403-413.
- [ISO92] International Organization for Standardization, Estelle Tutorial, October 26, 1992, ISO9074, Ammendment 1, Annex D, 55 pages.
- [KUUL91] I. Kuuluvainen, et al., "The Action-State Diagram: A Compact Finite State Machine Representation For User Interfaces and Small Embedded Reactive Systems", *IEEE Transactions on Consumer Electronics*, August 1991, pp. 651-658.
- [SHAW92] A. Shaw, "Communicating Real-Time State Machines", *IEEE Transactions on Software Engineering*, September 1992, pp. 805-816.
- [SIJE92] R. Sijelmassi and R. J. Linn, "Guidelines for using Estelle to specify OSI services and protocols", *Computer Networks*, Vol. 23, No. 5, pp. 343-362.

D.2 Petri Nets

- [BALB92] G. Balbo, et al., "An Example of Modeling and Evaluation of a Concurrent Program Using Colored Stochastic Petri Nets: Lamport's Fast Mutual Exclusion Algorithm", *IEEE Transactions on Parallel and Distributed Systems*, March 1992, pp. 221-239.
- [BERT91] B. Berthomieu and M. Diaz, "Modeling and Verification of Time Dependent Systems Using Time Petri Nets", *IEEE Transactions on Software Engineering*, March 1991, pp. 259-273.
- [CHIO93] G. Chiola, et al., "Generalized Stochastic Petri Nets: A Defintion at the Net Level and Its Implications", *IEEE Transactions on Software Engineering*, February 1993, pp. 89-107.

- [DUGG88] J. Duggan and J. Browne, "ESPNET: expert-system-based simulator of Petri nets", *IEE Proceedings*, Vol. 35, Pt. D, No. 4, July 1988, pp. 239-247.
- [GHEZ91] C. Ghezzi, et al., "A Unified High-Level Petri Net Formalism for Time-Critical Systems", *IEEE Transactions on Software Engineering*, February 1991, pp. 160-172.
- [LAUS88] G. Lausen, "Modeling and Analysis of the Behavior of Information Systems", *IEEE Transactions on Software Engineering*, November 1988, pp. 1610-1620.
- [MICO90] A. Micovsky, et al., "TORA: A Petri Net Based Tool for Rapid Prototyping of FMS Control Systems and its Application to Assembly", *Computers in Industry*, 15(4) (1990), pp. 279-292.
- [MURA84] T. Murata, "Modeling and Analysis of Concurrent Systems", *Handbook of Software Engineering*, (Vick and Ramamoorthy, editors), Van Nostrand Reinhold, New York, 1984, pp. 49-63.
- [MURA89] T. Murata, "Petri Nets: Properties, Analysis, and Applications", *Proceedings of the IEEE*, April 1989, pp. 541-580.
- [PAPE92] Y. Papelis and T. Casavant, "Specification and Analysis of Parallel/Distributed Software and Systems by Petri Nets With Transition Enabling Functions", *IEEE Transactions on Software Engineering*, March 1992, pp. 252-261.
- [TAQI92] A.A.Q. Taqi, et al., "A comparative study between Petri Net and SLAM", *Simulation*, November 1992, pp. 339-344.
- [WILL90] R. G. Willson and Bruce Krogh, "Petri Net Tools for the Specification and Analysis of Discrete Controllers", *IEEE Transactions on Software Engineering*, January 1990, pp. 39-50.
- [YAO89] Y. Yao, "An Approach to Formal Specification and Analysis for Time Performance of the Concurrent Real Time System (RTEXS)", *Computers in Industry*, 12(1989), pp. 347-354.

D.3 Temporal Ordering

- [BIEM86] F. Biemans and P. Blonk, "On the Formal Specification

and Verification of CIM Architectures Using LOTOS", *Computers in Industry*, (7) (1986), pp. 491-504.

- [ISO87] International Organization for Standardization, LOTOS -- A Formal Description Technique Based on the Temporal Ordering of Observational Behavior, July 20, 1987, 126 pages.
- [ISO92] International Organization for Standardization, LOTOS -- A Formal Description Technique Based on the Temporal Ordering of Observational Behavior, Amendment 1:G-LOTOS, April 30, 1992, 111 pages.
- [KARA91] G. Karam and R. Buhr, "Temporal Logic-Based Deadlock Analysis For Ada", *IEEE Transactions on Software Engineering*, October 1991, pp. 1109-1125.
- [LOGR88] L. Logrippo, et al., "An Interpreter for LOTOS, A Specification Language for Distributed Systems", *Software--Practice and Experience*, April 1988, pp. 365-385.
- [MUNS91] H. B. Munster, LOTOS specification of the MAA standard, with an evaluation of LOTOS, National Physical Laboratory, Report DITC 191/91, September 1991, 87 pages.
- [SIST91] R. Sisto, et al., "A Protocol for Multirendezvous of LOTOS Processes", *IEEE Transactions on Computers*, April 1991, pp. 437-446.

D.4 Other Relevant Formalisms

- [DILL90D] A. Diller, Z -- An Introduction To Formal Methods, John Wiley and Sons, New York, 1990, 309 pages.
- [GERB92] R. Gerber and I. Lee, "A Layered Approach to Automating the Verification of Real-Time Systems", *IEEE Transactions on Software Engineering*, September 1992, pp. 768-784.
- [HOAR85] C.A.R. Hoare, Communicating Sequential Processes, Prentice-Hall, Englewood Cliffs, N.J., 1985, 256 pages.
- [LAMP89] L. Lamport, "A Simple Approach To Specifying Concurrent Systems", *Communications of the ACM*, January 1989, pp. 32-45.

[POTT91] B. Potter, et al., An Introduction to Formal Specification and Z, Prentice-Hall, Englewood Cliffs, N.J., 1991, 304 pages.

E. Dynamic Modeling Approaches

[MAY87] P. J. Mayhew and P. A. Dearnley, "An Alternative Prototyping Classification", *The Computer Journal*, Vol. 30, No. 6, 1987, pp. 481-484.

[ZEIG84] B. P. Zeigler, "Theory and Application of Modeling and Simulation: A Software Engineering Perspective", Handbook of Software Engineering, (Vick and Ramamoorthy, editors), Van Nostrand Reinhold, New York, 1984, pp. 1-25.

E.1 Executable Specifications

[HULL86] M. E. C. Hull, "Implementations of the CSP Notation for Concurrent Systems", *The Computer Journal*, Vol. 29, No.6, 1986, pp. 500-505.

[LEE91] S. Lee and S. Sluzier, "An Executable Language For Modeling Simple Behavior", *IEEE Transactions on Software Engineering*, June 1991, pp. 527-543.

[NOTA92] G. Nota and G. Pacini, "Querying of Executable Software Specifications", *IEEE Transactions on Software Engineering*, August 1992, pp. 705-716.

[VALE93] A. Valenzano, et al., "Rapid Prototyping of Protocols from LOTOS Specifications", *Software--Practice and Experience*, January 1993, pp. 31-54.

[ZAVE82] P. Zave, "An Operational Approach to Requirements Specification for Embedded Systems", *IEEE Transactions on Software Engineering*, May 1982, pp. 250-269.

[ZAVE86] P. Zave and W. Schell, "Salient Features of an Executable Specification Language and Its Environment", *IEEE Transactions on Software Engineering*, February 1986, pp. 312-325.

E.2 Prototyping

[BERE84] W. E. Beregi, "Architectural prototyping in the software engineering environment", *IBM Systems Journal*, Vol. 23, No. 1, 1984, pp. 4-18.

- [BROW88] D. W. Brown, et al., "Software Specification And Prototyping Technologies", *AT&T Technical Journal*, July/August 1988, pp. 33-45.
- [CAME91] E. J. Cameron, et al., "The L.0 Language and Environment for Protocol Simulation and Prototyping", *IEEE Transactions on Computers*, April 1991, pp. 562-570.
- [CHOP90] C. Choppy and S. Kaplan, "Mixing Abstract And Concrete Modules: Specification, Development And Prototyping", Proceedings of the 12th International Conference On Software Engineering, 1990, pp. 173-184.
- [CHU87] W. W. Chu, et al., "Testbed-Based Validation of Design Techniques for Reliable Distributed Real-Time Systems", *Proceedings of the IEEE*, May 1987, pp. 649-667.
- [HARD88] B. Harding, "Executable modeling aids design of real-time embedded systems", *Computer Design*, December 1988, pp. 48-49.
- [LUQI92] Luqi, "Computer-Aided Prototyping For A Command-And-Control System Using CAPS", *IEEE* January 1992, pp. 56-67.
- [SAHR92] A.E.K. Sahraoui and N. Ould-Kaddour, "Control Software Prototyping", *Computers in Industry*, 20 (1992), pp. 327-334.

E.3 Simulation

- [DEME91] E. C. DeMeter and M. P. Deisenroth, "GIBSS: A model specification framework for multi-stage, manufacturing system design", *Simulation*, June 1991, pp. 413-421.
- [FINN92] A. Finn, et al., "Simulation of multiple access protocols for real-time control", *Simulation*, February 1992, pp. 123-130.
- [OZDE93] N. E. Ozdemirel and G. T. Mackulak, "A Generic Simulation Module Architecture Based on Clustering Group Technology Model Codings", *Simulation*, June 1993, pp. 421-433.
- [PARR92] G. Parr and P. Bielkowicz, "Layered simulation of Bridge protocols for Multi-LAN Ethernet Communication Systems", *Simulation*, February 1992, pp. 109-122.

- [PIDD92] M. Pidd, "Guidelines for the design of data driven generic simulators for specific domains", *Simulation*, October, 1992, pp. 237-243.
- [ROSE92] R. C. Rosenberg, et al., "Extendible simulation software for dynamic systems", *Simulation*, March 1992, pp. 175-183.
- [SAKT92] S. Sakthivel and R. Agarwal, "Knowledge-based model construction for simulating information systems", *Simulation*, October 1992, pp. 223-236.
- [ZEIG87] B. P. Zeigler, "Hierarchical, modular discrete-event modelling in an object-oriented environment", *Simulation*, November 1987, pp. 219-230.

E.4 Hybrid Approaches

- [BAGR91] R. J. Bagrodia and C. Shen, "MIDAS: Integrated Design and Simulation of Distributed Systems", *IEEE Transactions on Software Engineering*, October 1991, pp. 1042-1058.
- [BALD91] M. Baldassari, et al., "PROTOB: an Object-oriented CASE Tool for Modelling and Prototyping Distributed Systems", *Software--Practice and Experience*, August 1991, pp. 823-844.
- [ESTR86] G. Estrin, et al., "SARA (System ARCHitects Apprentice): Modeling, Analysis, and Simulation Support for Design of Concurrent Systems", *IEEE Transactions on Software Engineering*, February 1986, pp. 293-311.

F. Languages

- [BUDG85] D. Budgen, "Combining MASCOT with Modula-2 to aid the Engineering of Real-Time Systems", *Software--Practice and Experience*, August 1985, pp. 767-793.
- [DOTA91] Y. Dotan and B. Arazi, "Using Flat Concurrent Prolog in System Modeling", *IEEE Transactions on Software Engineering*, June 1991, pp. 493-512.
- [HANS87] P. B. Hansen, "Joyce -- A Programming Language for Distributed Systems", *Software--Practice and Experience*, January 1987, pp. 29-50.

- [ISHI92] Y. Ishikawa, et al., "An Object-Oriented Real-Time Programming Language", *IEEE Computer*, October 1992, pp. 66-73.
- [MEYE88] B. Meyer, Object-oriented Software Construction, Prentice-Hall, New York, 1988, 534 pages.
- [ROSE91] D. S. Rosenblum, "Specifying Concurrent Systems with TSL", *IEEE Software*, May 1991, pp. 52-61.
- [STEU84] H. U. Steusloff, "Advanced Real-Time Languages for Distributed Industrial Process Control", *IEEE Computer*, February 1984, pp. 37-46.
- [WISE93] M. J. Wise, "Experience with PMS-Prolog: a Distributed, Coarse-grain-parallel Prolog with Processes, Modules and Streams", *Software--Practice and Experience*, February 1993, pp. 151-175.

G. Other Related References

- [ABRA92] M. Abrams, et al., "Chitra: Visual Analysis of Parallel and Distributed Programs in the Time, Event, and Frequency Domains", *IEEE Transactions on Parallel and Distributed Systems*, November 1992, pp. 672-685.
- [CARD91] S. Cardenas-Garcia and M. Zelkowitz, "A Management Tool For Evaluation of Software Designs", *IEEE Transactions on Software Engineering*, September 1991, pp. 961-971.
- [DILL90G] L. K. Dillon, "Verifying General Safety Properties of Ada Tasking Programs", *IEEE Transactions on Software Engineering*, January 1990, pp. 51-63.
- [WANG93] Y. Wang and D. L. Parnas, "Simulating the Behavior of Software Modules by Trace Rewriting", Proceedings of the 15th International Conference on Software Engineering, May 1993, pp. 14-23.