AD-A105 459 MARYLAND UNIV COLLEGE PARK DEPT OF COMPUTER SCIENCE F/G 9/2
AN OPERATIONAL APPROACH TO REQUIREMENTS SPECIFICATION FOR EMBED--ETC(U)
AUG 81 P ZAVE F49620-80-C-0001
UNCLASSIFIED AFOSR-TR-81-0662 NL

1 of 1
AD
A105459

END
DATE
FILMED
0 81
DTIC

10 bs

AN OPERATIONAL APPROACH TO REQUIREMENTS SPECIFICATION FOR EMBEDDED SYSTEMS

Pamela Zave

Aug 1981

F49620-80-C-0001

Abstract The approach to requirements specification for embedded systems described in this paper is called "operational" because a requirements specification is an executable model of the proposed system interacting with its environment. The approach is embodied by the language PAISLey, which is motivated and defined herein. Embedded systems are characterized by asynchronous parallelism, even at the requirements level; PAISLey specifications are constructed of interacting processes so that this can be represented directly. Embedded systems are also characterized by urgent performance requirements, and PAISLey offers a formal, but intuitive, treatment of performance.

Index Terms requirements analysis and specification, embedded (real-time) systems, system design and specification, simulation models, distributed processing, applicative programming

8 0 45

AN OPERATIONAL APPROACH TO REQUIREMENTS SPECIFICATION FOR EMBEDDED SYSTEMS

0. INTRODUCTION

Recently the study of system requirements has emerged as a major area of research in software engineering. It has become clear that the stated requirements for a system have tremendous impact on the quality and usefulness of the ultimate product, and on the efficiency and manageability of its development. Yet, despite their leverage, relatively little is known about deriving and specifying good sets of requirements.

At the same time, the prominence of "embedded" (roughly equivalent to "real-time") systems has been increasing, due largely to hardware advances which have made them feasible for a broader category of applications than ever before. We will argue that embedded systems are characterized by the urgency of their performance requirements; to the extent that all computer systems would benefit from the ability to state and satisfy precise performance requirements, knowledge of embedded systems can be useful to developers of all types of system.

This paper presents a new approach to the problem of specifying the requirements for embedded systems. It offers a substantial increase in formality, expressive power, and potential for automation over the current widely known requirements technologies.

1. THE REQUIREMENTS PROBLEM

1.1. The role of requirements in the system life cycle

The development of a computer system begins with the perception of a need for it. During the requirements phase, analysts should arrive at a deep understanding of that need and propose a system to fill it. The product of the requirements phase is the requirements specification, which plays a unique and crucial role in the rest of development. It states what system is to be developed, at what costs, and under what constraints.

The project cannot be a complete success unless the requirements have the informed consent of everyone who will be involved, including members of the development organization (designers, programmers, and managers), the originating organization (managers or salespeople who determine the cost and value of the system), and the ultimate users of the system. This consensus can only be achieved through feedback and negotiation, with preliminary versions of the requirements specification being the major vehicle of communication.

During design and implementation, the requirements specification defines the "top" for top-down design, and the product toward which management effort is aimed. At the end of development, it is the standard against which the system is compared for success or failure, acceptance or rejection.

Requirements are often neglected, for reasons that are all too familiar: lack of awareness of their importance (which is disappearing), lack of useful requirements analysis and specification techniques, and natural reluctance to incur costs and delays at the beginning of a project. Yet the consequences of this shortsightedness, which include cancelled projects or unprofitable products, unhappy users, chaotically structured systems, budget and schedule overruns as endless changes are made, and even lawsuits, are so serious that no one involved in software engineering can afford to ignore them. Other introductions to the role of requirements in system development can be found in [Boehm 76], [Bell & Thayer 76], [Ross & Schoman 77], [Yeh _et al_. 80], [Heninger 79], [Balzer & Goldman 79], and [Davis & Rauscher 79].

It should be noted that even with the most optimistic view of current progress on requirements analysis and specification, in which problems of communication and complexity can be solved, certain other problems will remain very difficult to deal with. One is that vital decisions must be based on forecasts of costs and even feasibility, while such forecasting is perhaps the weakest point of our software technology. Another is that the requirements are constantly changing, even as we try to write them down. And systems that are used evolve continually throughout their lifetimes ([Belady & Lehman 79], [Lehman 80]), creating "maintenance" costs which may eventually dwarf those of initial development.

As consciousness of the economic and technical importance of evolution in the system life cycle grows, we may develop a new concept of the life cycle based on iterated (re)developments, large and small, as in [Conn 80]. In such a model, the requirements specification will evolve with the system, serving throughout its life as definition, documentation, and contract. Needless to say, this expanded role will place even greater demands on the quality and modifiability of our requirements specifications.

1.2. Goals for requirements specifications

Progress in software engineering has almost always been made from the bottom up: from machine language to axiomatic specifications, for example, we have proceeded first by learning to do something and then by understanding it well enough to find suitable abstractions of it. This paper takes the same approach to requirements. It seems unlikely that we will find really effective techniques for requirements analysis before we know how to write a good requirements specification recording the results of that analysis. Therefore we *will* concentrate on specification techniques (although useful results on specification cannot help but suggest analytic methods and principles).

Goals for requirements specifications have been examined by many of the cited authors, and are discussed at more length in [Zave 80a]. In short, the things we do with requirements specifications are: (1) use them as vehicles for communication, (2) change them, (3) use them to constrain target systems, and (4) use them to accept or reject final products.

For (1) and (2) they must be understandable and modifiable. ıor (3) they must be precise, unambiguous, internally consistent, and complete. They should also be minimal, i.e. define the smallest set of properties that will satisfy the users and originators. Otherwise the specification may over-constrain the target system, so that some of the best solutions to design problems are unnecessarily excluded. For (4) they should be formally manipulable (if verification is to be used) or testable (if acceptance testing is to be used).

The remainder of this paper is concerned with a requirements specification approach (and language) that promises to help us achieve many of these goals. It is also somewhat specialized for a particular class of systems, namely . . . .

2. EMBEDDED SYSTEMS

Common examples of embedded systems are industrial process-control systems, flight-guidance systems, switching systems, patient-monitoring systems, radar tracking systems, ballistic-missile-defense systems, and data-acquisition systems for experimental equipment. The class of embedded systems is an important one, partly because it already includes some of our oldest and most complex computer applications, and partly because it is expanding rapidly in volume and variety as a result of the microprocessor revolution.

2.1. What makes a system "embedded"?

The term "embedded" was popularized by the U.S. Department of Defense in conjunction with its common language (Ada) development project. "Embedded" refers to the fact that these systems are embedded in larger systems whose primary purposes are not computation, but this is actually true of any useful computer system. A payroll program, for instance, is an essential part of a business organization, which is a system whose primary purpose is selling products at a profit.

The common concept that unites the systems we choose to call "embedded" is process control: providing continual feedback to an unintelligent environment. This "theme" is easily recognized in flight-guidance systems and switching systems; even in a patient-monitoring system, sick patients are not exercising their intelligence in interacting with the system, and nurses can be viewed as providing a mechanical extension to the system's feedback loop.

The continual demands of an unintelligent environment cause these systems to have relatively rigid and urgent performance requirements, such as real-time response requirements and "fail-safe" reliability requirements. It seems that this emphasis on performance requirements is what really characterizes embedded systems, and causes us to be more aware of their roles in their environments than we are for other types of system.[1]

Figure 1 shows an informal classification of systems, based on properties that show up at the requirements level. Requirements for "support systems" are generally much less definite than requirements for applications systems. And while the performance requirements for embedded systems may be couched in absolutes, the performance requirements for support systems will be relative to resources and resource utilization, and the performance requirements for data-processing systems will be relative to load, resources, and psychological factors. The most complex systems, such as nationwide airline-reservation systems, should probably be viewed as having subsystems of all three types.

2.2. The special problems of embedded systems

The special nature of embedded systems exacerbates many software engineering problems, and thus demands particular attention even during the requirements phase.

Few organizations have logged as much experience with embedded systems as the Department of Defense, which spends 56 per cent of its approximately 3 billion dollar annual software budget on them ([Fisher 78]). Here is a pointed summary of that experience:

> Embedded computer software often exhibits characteristics that are strikingly different from those of other computer applications. The programs are frequently large (50,000 to 100,000 lines of code) and long-lived (10 to 15 years). Personnel turnover is rapid, typically two years. Outputs are not just data, but also control signals. Change is continuous because of evolving system requirements--annual revisions are often of the same magnitude as the original development ([Fisher 78]).

Clearly coping with complexity and change will not be easier in the domain of embedded systems.

In addition to the performance requirements, which have already been established as a major distinguishing factor, embedded systems are especially likely to have stringent resource requirements. These are requirements on the resources, mainly physical in this case, from which the system is constructed. This is because embedded systems are often installed in places (such as satellites) where their weight, volume, or power consumption must be limited, or where temperature, humidity, pressure, and other factors cannot be as carefully controlled as in the traditional machine room.

The interface between an embedded system and its environment tends to be complex, asynchronous, highly parallel, and distributed. This is another direct result of the "process control" concept, because the environment is likely to consist of a number of objects which interact with the system and each other in asynchronous parallel. Furthermore, it is probably the complexity of the environment that necessitates computer support in the first place (consider an air-traffic-control system)! This characteristic makes the requirements difficult to specify in a way that is both precise and comprehensible.

Finally, embedded systems can be extraordinarily hard to test. The complexity of the system/environment interface is one obstacle, and the fact that these programs often <u>cannot</u> be tested in their operational environments is another. It is not feasible to test flight-guidance software by flying with it, nor to test ballistic-missile-defense software under battle conditions.

3. AN "OPERATIONAL" APPROACH

The approach taken in this paper is to specify the requirements for an embedded system with an explicit model of the proposed system interacting with an explicit model of the system's environment. Both submodels consist of sets of asynchronously interacting digital processes, although some of the processes in the environment model may represent discrete simulations of nondigital objects such as people or machines. The entire model is executable, and the internal computations of the processes are specified in an applicative language.

We call this an "operational" approach because the emphasis on constructing an operating model of the system functioning in its environment provides its primary flavor. It has been embodied in a Specification Language which, since it is based on the ideas above and is therefore Process-oriented, Applicative, and Interpretable (executable), is named PAISLey.

In the remainder of this section, the basic ideas behind PAISLey will be explained, illustrated, and justified in detail. Section 4 addresses apparent disadvantages of an operational approach, and Section 5 defines PAISLey.

3.1. Explicit modeling of the environment

Figure 2 shows a partial requirements model for a simple patient-monitoring system. The "patient", "nurse", and "doctor" processes are all digital simulations of these natural objects. They represent, of course, only the roles played by these people with respect to patient-monitoring. The "patient" process also models a sensor attached to the patient (this model is only partial because the complete one would have multiple patients, sensors, terminals, etc.).

The "reader" process reads sensor data at specified intervals of real time, sending a warning to the terminal if the reading is so implausible as to suggest sensor malfunction. The "monitor" process checks the reading against medical safety criteria, sending a warning if it falls outside a safe range. Sensor data also goes to the "database" process, which responds to queries from the terminal and also purges old data to maintain itself at a reasonable size. Many parameters, such as sensor-reading frequencies and safe ranges, can be set by doctors and nurses.

The boundary between the environment and the proposed system is determined simply by which parts are "givens" for the contractor and which parts must be supplied by the contractor. The boundary is arbitrary (from a technical viewpoint), and is not even part of the formal PAISLey specification.

Including an explicit model of the environment has several advantages for requirements specification. The reason that the interface between an embedded system and its environment is complex, asynchronous, highly parallel, and distributed is that it consists of interactions among a number of objects which exist in parallel, at different places, and are not synchronized with one another. Organizing these interactions around the objects (processes) which take part in them is an effective way to decompose this sort of complexity. Furthermore, assumptions and expectations on both sides of the boundary can be documented. The result is a specification which is far more precise and yet comprehensible than could be obtained by treating either side of the interface as a "black box", which is what happens when the environment is not modeled.

Another reason for having an environment model is that the environment (when construed broadly enough) is the source of all changes to the system. Modeling it is therefore a promising way to anticipate changes and enhance the modifiability of both specification and target system.

The final advantage of specifying the environment is that many performance constraints are most naturally attache there. The patient-monitoring system has a real-time response requirement on database queries, which is neatly expressed as a time limit on the component of the terminal specification which waits for the response after sending a query. The system must also be able to handle a certain load. Since this load is completely determined by the numbers of sensors and terminals and the rates at which they create work for the system, it is most directly specified by a model of those peripherals.

The other significant aspect of constructing an environment model is that it is a valuable tool for requirements _analysis_, as well as specification. In fact, the best way to analyze requirements may be to start with the environment model, and work "outside-in" to a proposed system which supports a desirable mode of operation in the environment. The extreme case is automation of an existing manual system--in the absence of changes to existing procedures, the requirements can be derived simply by modeling the current operation, and drawing a boundary to distinguish the automatable part! [Yeh _et_ _al_. 79a] and [Yeh _et_ _al_. 79b] both discuss "conceptual models", which are models of system environments constructed for the purpose of requirements analysis.

In the patient-monitoring system, since only the "_patient_" and "_crt-terminal_" processes interact directly with the proposed computer system, only these are necessary for precise specification of the system interface. The "_nurse_" and "_doctor_" processes appear strictly as vehicles for requirements analysis. Wondering how doctors and nurses interact leads the analyst to ask which kinds of information a doctor expects to get from a nurse on duty, and which kinds he would like to find in the database. Wondering how nurses interact with patients and the display leads the analyst to ask how the display screen should be allocated to medical histories versus emergency messages, how often warnings concerning an ongoing crisis need be displayed, and whether information from the monitoring system is needed at the patient's bedside. These questions are never asked (or answered) in the numerous treatments of patient-monitoring systems appearing in the requirements literature.

Even if the analysts can achieve understanding of the requirements in some other way, early concentration on the environment may lead to better communication with users (who are much more interested in their environment than your system), and more open-minded problem-solving, unbiased by preconceived notions or similar systems the analysts have worked on.

3.2. Processes

Another key feature of the operational approach is that the primary units of specification are processes. A process is a simple, abstract representation of autonomous (distributed) digital computation. It is specified by supplying a "state space", or set of all possible states, and a "successor function"[2] on that state space which defines the successor state for each state. It goes through an infinite sequence of states (although a "halting" process can be specified by having it go into a distinguished "halted" state which it will never leave), asynchronously with respect to all other processes (Figure 3).

A process is cyclic, with its successor function describing its natural cycle. The natural cycle of a process simulating a sick patient, for instance, would be a single step of the discrete simulation algorithm. The successor function of such a process might be declared as:

patient-cycle: PATIENT-STATE —> PATIENT-STATE,

where the set "PATIENT-STATE", which is its domain and range and also the state space of the process, contains values encoding possible states of the patient between simulation steps. The natural cycle of the "doctor" process might be to take one action, either asking one question of a nurse, giving one order to a nurse, or taking part in one transaction with the patient-monitoring system.

There can be no question about the generality of processes. They were originally used as abstractions of concurrent activities within multiprogramming systems ([Horning & Randell 73]), and many recent articles have shown that they can be used to represent I/O devices, data modules, tasks, monitors, buffers, or any other identifiable structure within a computer system (e.g. [Hoare 78], [Brinch Hansen 78], [Mao & Yeh 80]). Process-based models of computation have been the focus of extensive theoretical work and the language Smalltalk ([Ingalls 78]). Our varied examples are persuasive evidence that the notion of digital simulation of nondigital objects is similarly powerful in describing the environments of computer systems.

The appropriateness of using processes to specify requirements for embedded systems is based on our observation that in these systems asynchronous parallelism--among environment objects, between environment objects and the system, and within the system (if only for reasons of performance)—occurs naturally at the requirements level. One happy result of recognizing that parallelism is environment specifications which should be highly intuitive, even to naive users. This is because the specifications are populated by identifiable models of the same autonomous, interacting objects from which the real world is made.

Perhaps the best way to appreciate processes is to consider the alternatives: representations of processing used in other requirements languages. The one most commonly found in requirements documents is data access or "dataflow". Data access graphs show major system functions, and identify the data structures which are their inputs and outputs (e.g. Figure 4). Data access is the basis of PSL/PSA ([Teichroew & Hershey 77]) and SADT ([Ross 77], [Ross & Schoman 77]), and has probably been rediscovered thousands of times by isolated requirements-writers.

Data access may be adequate for many data-processing systems, such as the one depicted in Figure 4. This is because major subfunctions ("check-inventory", "send-invoice") are implemented as major subprograms, and subprograms are invoked in some implicitly understood sequence, whenever their input files are ready. Data access is seriously inadequate for embedded systems, however, because control is all-important in embedded systems and must be represented in any intelligible model. If a data-access graph is mistakenly interpreted as representing control as well, the concepts of control expressed will be simplistic and misleading.

In Figure 4, for instance, the following problems would arise if control were implied: (1) A distinction must be made between inputs which are always present (such as the "INVENTORY" database) and inputs which invoke a function whenever a new instance appears (such as "PURCHASE-ORDER"). The situation is even more complex when there is an input value (such as the current output of a sensor attached to a hospital patient) which is always available, but only read at certain real-time intervals (and the interval itself is a variable stored in some system database). (2) Functions (such as "process-account-order" and "process-payment") may have to be executed concurrently to meet performance requirements, in which case they must synchronize their uses of shared resources or databases (such as "ACCOUNTS"). (3) Functions may no longer execute in a predefined sequence (because of simultaneous access from multiple terminals, the need for internal housekeeping, etc.), and so a complex interplay of events and states must be anticipated.

The control arrow in SADT adds an explicit representation of control to data-access graphs (an illuminating discussion of its significance can be found in [Ross 77]), but its informality prevents it from being precise or expressive enough for embedded systems. Processes and their interactions, on the other hand, are well-suited to the task of specifying complex control, as would be expected from their historical origins in the specification of operating systems.

Other representations of processing appearing in requirements languages are stimulus-response paths in RSL ([Bell et al. 77], [Alford 77], [Davis & Vick 77]) and finite state machines ([Heninger 79], [Davis & Rauscher 79]). A finite state machine is very much like a single process--permitting no explicit parallelism, decomposition of complexity, nor modeling of the environment.

Stimulus-response paths (e.g. Figure 5) represent sequencing, control, and parallelism explicitly, and do make it possible to decompose the requirements. The "R-net" in Figure 5 shows parallelism between "STORE_FACTOR_DATA" and "EXAMINE_FACTORS", and is only one of several R-nets specifying the entire system. The fundamental differences between PAISLey and RSL appear to be as follows: (1) The PAISLey representation emphasizes the cyclic nature of system components, while the RSL representation emphasizes sequences. Both are obviously useful ways of characterizing embedded systems, and only time will tell if one is superior to the other in a majority of instances. (2) The PAISLey notation integrates data, processing, and control in a unified whole, while in RSL the various concepts are separated—R-nets for control, PSL/PSA-like notation for data-access properties, etc. We believe that the unified approach will ultimately prove stronger in terms of comprehensibility, modifiability, and ability to determine internal consistency.

3.3. Executability

In the operational approach, requirements specifications are executable. This means that, under interpretation, the specification becomes a simulation model generating behaviors of the specified system.

It is of vital importance to be able to interpret specifications regardless of their level of abstraction. Not only are requirements by their nature abstract in many respects, but they must also be developed by successive refinements of understanding, each version of which should benefit from this facility. We will defer until 3.4 a discussion of how this can be done, and only deal here with the advantages of doing so.

Executability is a powerful tool for understanding a specification in all its ramifications. An executable specification can be tested and debugged by the analysts who wrote it. Its behavior can be validated by users in the course of demonstrations. If developed in enough detail, it can even be released on a small scale as a prototype of the proposed system.[3]

The ability to test is no panacea, as must be obvious from the literature on program testing—and with embedded system specifications there is the additional complication that any test must choose one of many relative-rate-dependent process execution sequences. Nevertheless, the problems inherent in testing programs have never caused us to give up on it, and requirements testing, once established in common practice, might seem likewise indispensable.

Furthermore, an executable requirements model can continue to be useful after the requirements phase. The environment part of the model can be used as a test bed during system development, which will be particularly valuable for embedded systems because of the aforementioned difficulties of testing them "in the field" (in fact, it is almost always necessary to write an environment simulator for exactly this purpose). The model of the proposed system can be used to generate sample behaviors for acceptance testing.

It is also possible to attach performance constraints in such a way that they can be simulated along with the functional requirements, and this is done in PAISLey. Simulation can then be used to predict performance where it is too complex to determine analytically. This type of simulation is an important feature of SREM, the integrated set of tools by which RSL is supported.

There is a final, critically important, advantage of executability that has nothing to do with testing or simulation. It is that the demands of executability impose a coherence and discipline--because the parts of a specification must "fit together" in a very strong sense--that could scarcely be obtained in any other way. If an executable requirements specification is shown to be internally consistent, that means it will continue to generate behaviors without ever halting, deadlocking, or going into an undefined state. In other words, it is guaranteed to be a valid specification of <u>some</u> system interacting with <u>some</u> environment. Clearly this is the utmost that any formally defined notion of internal consistency could do for us, since deciding whether they are the <u>right</u> system and environment is a matter of validation by the originator/user, or verification of consistency with externally defined axioms of correctness.

3.4. Specification in an applicative language

Within a process, computation (i.e. the successor function of the process) is specified using a purely applicative language. "Applicative" (or "functional") languages are those based on side-effect-free evaluation of expressions formed from constants, formal parameters, functions, and functional operators ("combining forms" for functions, such as composition). Well-known examples of applicative languages are the lambda calculus, pure LISP, and the functional programming systems of [Backus 78].

3.4.1. Advantages of applicative languages

Applicative languages are currently receiving a great deal of favorable attention because of their numerous theoretical and practical advantages ([Backus 78], [Iverson 80], [Smoliar 80], [Friedman & Wise 77], [Friedman & Wise 78a], [Friedman & Wise 78b], [Friedman & Wise 79], [Friedman & Wise 80], among others), most of which can be exploited in requirements specifications. To begin with, because applicative languages are interpretable, they support the executability property: processes are executed by repeatedly replacing their current states by successor states, and successor states are discovered by interpreting the applicative expressions which define them.

For purposes of high-level specification, the most important property of applicative languages is their tremendous powers of abstraction, i.e. of decision deferment. Consider, for instance, the functional expression "f[(g[y],h[z])]", which says that the function "g" is to be applied to the argument "y" and "h" is to be applied to "z" (the "[ ]" symbols denote function application or composition). Then "f" is to be applied to the values produced (the "( )" symbols are used to construct tuples of data). The expression says exactly what is needed to compute the desired value, but does not otherwise constrain the data, control, processor, or other resource structures used to do so. Are "g[y]" and "h[z]" evaluated sequentially or in parallel? In what data structures are their values stored? Perhaps the arguments "y" and "z" are even shipped off to special "g"- and "h"-processors, respectively, at different nodes of a network!

Furthermore, a primitive function has several interesting interpretations, all of which enable additional decompositions of complexity. A primitive function can represent a set of deferred decisions, to be made later by defining the function in terms of simpler primitives. It can also represent a mapping which will always remain nondeterministic from the perspective of the requirements model, because it depends on factors outside the scope of the model. For instance, in specifying a terminal we might declare a primitive function

```
think:  DISPLAY —> INPUT,
```

where "DISPLAY" is the set of all CRT screen images and "INPUT" is the set of all input lines, to represent the human user's thought processes. Finally, in PAISLey a primitive function can be an abstraction for an interprocess interaction (see below). Because of these many options, applicative languages have been used successfully to describe phenomena ranging in level of abstraction from digital hardware to distributed system requirements ([Fitzwater & Zave 77], [Smoliar 79]).

An interpreter for an abstract specification language makes expedient and non-functionally-significant decisions about such matters as control and space allocation. The only other thing needed for interpretation is some sort of implementation of functions and sets left primitive in the abstract specification. This can be done in many ways, perhaps the simplest of which is to choose values of primitive functions randomly or by default. Another way to interpret primitive functions is to display their arguments at a terminal and ask the analyst to supply a value, thereby creating an interactive testing system. In either case, the effect is to simulate the decisions which have been made, without interference from the decisions that haven´t been made.

Other advantages of applicative languages are that they are well-suited to formal manipulations such as verification, and may have great potential for efficient implementation (see [Backus 78]). In procedural languages, on the other hand, assignment statements thwart top-down thinking, complicate the formal semantics, and force memory to be referred to and accessed one word at a time.

3.4.2. PAISLey as an applicative language

PAISLey is not a purely applicative language because states in general, and process states in particular, are not applicative concepts. In [Zave 81] it is explained that, while many aspects of even embedded systems can be specified applicatively, the specification of most performance requirements, real-time interfaces with the environment, and certain resource requirements all necessitate the introduction of some non-applicative structure such as processes.

In fact, the process structure makes PAISLey specifications easier to write than typical large applicative programs. The system is decomposed into processes, and process computations are decomposed into cycles or steps, before applicative programming comes into use. The current state of a process "remembers" all its relevant history. Thus the applicative expressions which must be written are quite simple relative to the complexity of the system as a whole.

Since PAISLey is a blend of the applicative world and the non-applicative world of processes and states, the "seam" must be a smooth one. The two worlds meet at the mechanism for interprocess interaction, which is necessitated by the existence of processes but designed to fit smoothly into the applicative framework. Interactions take place through a set of three primitives called "exchange functions" which carry out the side-effect of asynchronous interaction, but look and behave locally (intraprocess) exactly like primitive functions. Exchange functions are defined and explained in 5.3. They are a unique mechanism which seems to fulfill our purposes very well, and also offers an interesting new perspective on asynchronous interaction mechanisms for distributed processes.

Applicative languages have, in some circles, a reputation for unreadability and general unsuitability for large-scale software engineering. We believe that this reputation is due to typelessness and recursion, neither of which is present in PAISLey.

Recursion is what purely applicative languages use to specify repetitive computation, and is analogous to looping (iteration) in procedural languages. Both are analogous to the repetitive application of a successor function to produce successive process states, which is how unbounded repetition is specified in PAISLey.

In most applicative languages, the only type of data object is the list or sequence, and all functions are applied to one list and produce one list. Since every function should be prepared to accept argument lists of any internal structure, there must be a distinguished "undefined" value produced whenever the internal structure of the argument is unsuited to the semantics of the function (as in [Backus 78])--and this mismatch must first be detected! Multiple arguments to or values from functions must be packaged in single lists, yet the existence of this substructure (or any other substructure, for that matter) cannot be explicitly acknowledged.

Of course, deliberate substructure in data items is ubiquitous, and it is common practice to document it with the use of data types. Furthermore, typing in a language provides a useful form of redundancy which is susceptible to automated checks of internal consistency.

In PAISLey non-primitive sets can be defined using set union ("A U B"), cross-product ("A x B"), enumeration ("{ ´true´, ´false´ }"), and parenthesization. The domain and range sets of every function, primitive or not, must be declared (although a function need not have arguments). The domain and range declarations can use arbitrary set expressions. Here are three example declarations:

```
f:  ---> A ;
g:  B x C ---> D U E ;
h:  S ---> T.
```

When a function is applied to arguments, their types must be consistent with the domain declaration of the function. Consistency can be defined with the assistance of coercion, however, so that the composition "h[g[f]]" is perfectly legal if the definitions:

```
A = B x C ;
S = INTEGERS;
D = { 0, 2, 4, 6, 8 };
E = { 1, 3, 5, 7, 9 }
```

have been made. This notion of typing provides all the documentation and redundancy desirable for engineering goals, without sacrificing any of the flexibility attributable to typelessness. All that it requires is the ability to compare any two set expressions for containment, which is easily done _given this particular language of set expressions_.

4. QUALMS ABOUT OPERATIONAL REQUIREMENTS

Despite the obvious advantages of operational requirements, one cannot help but have certain reservations about the idea. In this section we examine its apparent disadvantages.

4.1. _Encroaching on design_

Aren't operational specifications actually _design_ specifications rather than _requirements_ specifications? This question is often prompted by the precision, potential for detail, and executability of PAISLey specifications.

From a certain technical viewpoint, requirements should specify only the functional and performance properties of the proposed system. Design begins when resources are introduced. The physical resources from which the system is to be built must be managed so that performance goals are met. The human resources who will implement and maintain the system must be managed so that they are used effectively; this depends on skillful modularization of the code to be written. Adopting this as a definition of the boundary between requirements and design, a requirements specification does not stray into design if it avoids unnecessary management of resources and unnecessary structuring of code.

PAISLey enables the requirements analyst to do this. This point is illustrated copiously in [Zave 80a], using as an example the specification of a process-control system from [Zave & Yeh 81]. We will confine ourselves here to explaining why one aspect of a PAISLey specification, its process structure, does not over-constrain resource management or code structure.

Processes are virtual structures, and a specification is partitioned into processes on the basis of factors such as functionality, synchronization, and performance--all of which belong to the requirements level. Thus a requirements analyst should use as many processes as "make sense" to him. Only in design should there be concern about how a large number of virtual processes are to be realized on a smaller number of physical processors, perhaps through time-multiplexing with priority scheduling. And the processes, representing dynamic structure of the target system, say nothing about its static code structure at all.

This technical view of requirements is elegant and satisfying, but it is not the whole story. Pragmatically, a requirement is any property of the proposed system that is necessary to satisfy the originating organization of the acceptability of the system, and these properties may very well include decisions about resources. Use of a particular computer or software subsystem may be required because the originating organization already owns it, and management insists that it be used. The new system may have to interface with an existing computer system, and thus be compatible with its resource management policies. The capabilities of the proposed system may even have to be trimmed to fit the resources available. This is common with really large systems such as ballistic-missile-defense systems and massive database systems. Another example would be a system to monitor experimental equipment on a satellite. Since facilities must be shared with other experiments, the amount of memory in the on-board computer allocated to each experiment is an administrative decision which must be made (at least tentatively) before work on the individual projects can begin.

The reality is that a system develops through a hierarchy of decisions, each decision constraining those below it in the hierarchy. No system is developed in a political or economic vacuum, and almost no system performs its function without interfacing with any pre-existing computer system. Thus, even though resource decisions may be premature at the requirements level, any requirements language which is unable to record them will be terribly fragile, performing adequately only in the most idealized of situations.

PAISLey can record resource decisions because resource structures are like any other structures occurring in digital systems. They can definitely be specified if there is a general model of digital computation, which PAISLey offers. Hardware and software modules, for instance, can be specified as processes, and then included as part of the environment of the proposed system. This is a great strength of the operational approach--the promise of no unpleasant surprises when new applications, constraints, or economic contexts are encountered.

The ultimate test of whether or not a decision belongs in the requirements is whether or not the system could be constructed in any other way. This is illustrated in [Zave 80a], using the requirements from an early real-time simulation system ([Air Force 65]). It is shown that decisions about the communication network, timestamps, and the simulation time frame, while seemingly design decisions, are actually derivable during requirements analysis from feasibility considerations. The specification in PAISLey of each such decision is also described.

4.2. Too much precision

There can be little question that specifications written in PAISLey are too precise, and based on too many technical principles, for customers, end users, managers, and other untrained personnel to understand. At the same time, their rigor can be invaluable to the trained analysts who will write them (this is based on numerous experiences of being confronted by surprise with the vagueness of my own ideas about a system). Informal analysis must always come first, but we have not yet fully exploited the potential of formal languages for expressing approximate or incomplete knowledge and real-world concepts.

There is not really a conflict here, simply because nontechnical people do not have to use the same representations that the analysts do. Analysts can communicate with them using simplified diagrams and narrow views derived from the current PAISLey specification. A process diagram such as Figure 2, for instance, can have its interaction arrows labeled with the types o· data being transferred. If this is done, the diagram does not differ substantially in form or content from the ever-popular data-access graph!

The single most successful feature of SREM (RSL) and PSL/PSA is that specifications are stored in a database from which a variety of up-to-date reports can be generated automatically. We envision PAISLey's being installed in such a database, and hope that user-oriented reports and diagrams could likewise be produced by tools running on the current specification.

4.3. Interface with data-oriented specification techniques

Other researchers have investigated the problem of requirements for data-processing systems, using as a starting point for their formalisms data definition languages, i.e. languages originally developed to describe the "conceptual schemas" (abstract, virtual, semantic structures) of databases. The notion that a requirements model should be an explicit representation of the proposed system interacting with its environment has also been derived in this context. A philosophy of data-oriented modeling is presented in [Balzer & Goldman 79], while [Yeh _et al_. 79b], [Roussopoulos 79], [Mittermeir 80], [Smith & Smith 79], and [Goldman & Wile 80] exemplify it.

It is clear that a data-oriented technique is a more natural way than using PAISLey to develop requirements for data-processing systems. Yet data-processing systems cannot ignore performance and concurrency, and embedded systems cannot ignore data. Thus our goal is to view both process-oriented PAISLey specifications and data-oriented specifications as projections of the same underlying, all-encompassing model (Figure 6). In this view the two types of specification are compatible and complementary, so that analysts are free to use either or both (in parallel) as the application domain and phase of development suggest. In is argued in [Zave 80a], by describing the data definition facilities of PAISLey in traditional database terms, that the structure of PAISLey is not inconsistent with this goal.

5. THE PAISLEY LANGUAGE

In this section full details of PAISLey are presented, including a new mechanism for process interactions and specification of performance requirements. An LALR grammar for PAISLey in BNF form can be found in the Appendix.

5.1. Language philosophy

PAISLey is intended to be simple. In particular, only features which are directly associated with run-time semantics are included.

For production purposes the language must be supported by a system which, in addition to storing specification fragments and collecting them into executable configurations (not to mention providing tools for static analysis and report generation), offers such conveniences as scopes, versions, macros, parameters, libraries, meta-notations, etc. The current frenzy of research on "programming environments" makes it plain that the design of such an environment is not a trivial task, and should probably not be undertaken simultaneously with development of the specification semantics. Specifications prepared using any of the above features would be translated into PAISLey (as currently defined) before interpretation.

Stylistically, PAISLey follows APL in using distinct symbols for distinct operators (but has far fewer of them!). This leads to a concise notation in which essentially all words are user-chosen mnemonics. In this decision and the one above, we apply exactly the same philosophy as [Hoare 78].

One other important principle is that every operational structure must be realizable with a bounded amount of resources (time and space). There is a bounded number of processes, no process state can require an unbounded amount of storage, and no process step can require an unbounded amount of evaluation time. The purpose of this is performance, i.e. making it possible to design systems which are guaranteed to meet their performance requirements. Clearly if a computational path contains an unbounded loop, or may have to construct a data structure of unbounded size, no guarantee that it meets an absolute time constraint is possible. In PAISLey the only unbounded "structure" is the infinite succession of process steps of each process, and this one exception cannot be avoided.

Boundedness is enforced by requiring the sizes of system structures to be fixed. This gives the specification a static character which will greatly facilitate proofs of internal consistency, correctness, and other formal properties.

5.2. Sets, functions, processes, and systems

Statements in PAISLey are delimited by semicolons, and comments are enclosed in double quotation marks.

Names are typed for greater readability. The names of functions are always in lower-case letters, and the names of sets are always in upper-case letters (hyphens and integers may be used in *either*, but they must begin with alphabetic strings). Constants are either numbers, or strings enclosed in single quotation marks.

There are four kinds of statement: system declarations, function declarations, set definitions, and function definitions. Since a system is a fixed tuple of processes, we use the tuple-construction notation for a system declaration. A process is declared using a function application which applies its successor function to an expression evaluating to its initial process state. Thus a system consisting of four processes, three being terminals and the fourth being a shared database, would be declared as:

```
(terminal-1-cycle[blank-display],
 terminal-2-cycle[blank-display],
 terminal-3-cycle[blank-display],
 database-cycle[initial-database]
),
```

where the following domain-range declarations would be appropriate:

```
terminal-1-cycle:  DISPLAY ---> DISPLAY;

blank-display:  ---> DISPLAY;

. . .

database-cycle:  DATABASE ---> DATABASE;

initial-database:  ---> DATABASE.
```

Terminal processes have the contents of the current displays as their process states. Note that there is no explicit naming of processes or systems, as it is not needed for the run-time semantics.

Function declarations give <u>properties</u> of functions, i.e. specify their domains and ranges or their performance (see 5.4). All function declaration statements begin with the function name and a colon. Domain-range declarations (of which we have seen many) are mandatory for all functions except intrinsic ones. (This is because intrinsic functions are either intrinsically typed or may have to handle several different types in the same specification.) Domain-range declarations are optional for intrinsic functions, and performance declarations are optional for all functions.

When a function is nonprimitive (defined), its declarations may be redundant, because its properties may be deducible from its definition. Declarations of nonprimitive functions can and should be checked for consistency with their definitions.

Set definitions define set names in terms of *set expressions*, which use set union, cross-product (which has precedence over union), enumeration, and parenthesization (all but parenthesization shown in 3.4.2). Note that the size of all data structures is bounded, because all tuples (members of sets defined by cross-product) are of fixed size.

Function definitions define function names in terms of function expressions, and may use formal parameters to do so. The structure of the argument may be imitated in a formal parameter list, thus giving names to argument substructures. Here, for instance, are some possible beginnings for function definition statements:

```
new-func-1 = . . . ;

new-func-2[p] = . . . ;

new-func-3[(p,q)] = . . . ;
```

```
    new-func-4[(p,(q,(r,s)))] = . . . .
```

Now in a defining function expression (for all but "new-func-1") the argument´s first component can be referred to as "p", as an alternative to selectors such as "car" and "cdr" in LISP. Formal parameters have the same syntax as function names; the argument structure must, of course, agree with the function´s domain declaration.

Function expressions may use function names, formal parameters, constants, applications of functions to arguments, tuple construction, and conditional selection. Conditional selection (the "McCarthy conditional") has the syntax "/p1:f1, p2:f2, . . . ´true´:fn/" and evaluates to the value of the first functional expression "fi" such that the predicate (Boolean-valued functional expression) "pi" evaluates to "´true´". Note that there is no unbounded iteration, such as would be provided by "while . . . do . . .", nor is recursion allowed. Fixed iteration can be specified using composition. The result is that the number of primitive operations to evaluate any function, including a successor function, can be bounded *a priori*.

As a simple example, consider the following specification of the successor function of a process representing a CRT terminal:

```
    terminal-cycle:  DISPLAY ---> DISPLAY;

    terminal-cycle[d] = display
                            [display-and-transact
                                 [(d,think-of-request[d])]];

    think-of-request:  DISPLAY ---> REQUEST;

    display-and-transact:
         DISPLAY x REQUEST ---> DISPLAY x (RESPONSE U ERROR-MESSAGE);

    display-and-transact[(d,r)] = (display[(d,r)],transact[r]);

    transact:  REQUEST ---> RESPONSE U ERROR-MESSAGE;

    display:  DISPLAY x (REQUEST U RESPONSE U ERROR-MESSAGE) ---> DISPLAY.
```

The process handles one transaction per process step, reflecting both the request and the response in the display. The primitive function "display" can carry out scrolling or whatever other formatting is desired.

Even aiming for a minimum of conveniences, it is impossible to do without some feature for defining groups of nearly identical items. In PAISLey this is done at all levels using the same index notation, as seen in:

```
VECTOR = #1..10< x INTEGER >,
```

which defines members of the set "VECTOR" to be 10-tuples of integers. Index notation always denotes a sequence of the expression in angle-brackets, with the first symbol in the brackets used as the sequence delimiter. The integers after the "#" give the lower and upper bounds of the sequencing count. The only index notation without a delimiter symbol is the one for bounded functional composition (application), which makes "#1 .. 3 < func > [arg]" equivalent to "func[func[func[arg]]]".

In most cases what we want is a group of statements or expressions which differ slightly. This is done by operating on names, which are defined so that "syllables" (alphabetic substrings delimited by hyphens) are semantically meaningful. If the header for an index notation begins with a "syllable" before the "#", any syllable matching it in a name in the repeated expression will be replaced by successive integers from the lower bound to the upper bound.[4] Thus:

```
BIG-SET = J#1..3< U LITTLE-SET-J >
```

is equivalent to:

```
BIG-SET = LITTLE-SET-1 U LITTLE-SET-2 U LITTLE-SET-3,
```

and the system declaration:

```
(k#0..9999<,terminal-k-cycle[blank-display]>,
 database-cycle[initial-database]
)
```

creates a system with 10,000 terminal processes (an airline reservation system!), where the successor function of the thirteenth one is "terminal-12-cycle".

Index notation can even extend over groups of statements. Suppose we want our 10,000 terminals to be identical, except that some identification must be built into "transact", the primitive function whose elaboration will send to and receive from the central system. This can be done by making slight modifications to the terminal specification already given, as follows:

```
k#0..9999
< ; terminal-k-cycle: DISPLAY ---> DISPLAY;
    terminal-k-cycle[d] = display
                              [display-and-k-transact
                                   [(d,think-of-request[d])]];

    display-and-k-transact:
         DISPLAY x REQUEST ---> DISPLAY x (RESPONSE U ERROR-MESSAGE);

    display-and-k-transact[(d,r)] = (display[(d,r)],k-transact[r]);

    k-transact: REQUEST ---> RESPONSE U ERROR-MESSAGE;
>;
think-of-request: DISPLAY ---> REQUEST;
display: DISPLAY x (REQUEST U RESPONSE U ERROR-MESSAGE) ---> DISPLAY.
```

5.3. Asynchronous interactions

5.3.1. Definition of exchange functions

Asynchronous interactions between processes are specified using three types of primitive function known collectively as "exchange functions". An exchange function carries out two-way point-to-point mutually synchronized communication. It has one argument, which provides a value to be output, and always returns a value which was obtained as input. Thus within the process an exchange function looks like any other primitive function; it has, however, the side-effect of carrying out a process interaction. By making interaction primitives masquerade as functions, we achieve compatibility with applicative notation.

An exchange function whose evaluation has been initiated interacts by "matching" (to be explained) with another pending exchange function. The two exchange arguments and terminate, so that each returns as its value the argument of the other.

Each exchange function has two attributes to be specified, namely a type ("x", "xm", or "xr") and a channel (a user-chosen identifier which has the syntax of a function name). The exchange function with type "x" and channel "chan" is named "x-chan", the exchange function with type "xr" and channel "real-time" is named "xr-real-time", etc. Only exchange functions with the same channel can match with each other.

The "x" is the basic type of exchange function. It can match with any other pending exchange function on its channel, including another of type "x". If no other exchange is pending, it will wait until one is. If there are several pending match possibilities, a match will be chosen nondeterministically, with the proviso that there must be no lockout (a situation where a pending exchange waits indefinitely while its match opportunities are given to other, more recently evaluated, exchange functions).

Competitive situations occur in most systems. To enable succinct specification of them we have exchanges of type "xm", which behave exactly like "x"´s except that two "xm"´s on the same channel cannot match with each other. They can then compete to match with an exchange of some other type, as the examples will show.

Embedded systems typically have real-time interfaces, especially with the processes in their environments. To specify these we need a third type of exchange function, the "xr", which behaves like the others except that it will not wait to find a match. If evaluation of an "xr" is initiated and there is no other pending exchange on its channel, the "xr" terminates immediately without matching, returning its own argument as its value. It is always possible to determine whether or not an "xr" matched by giving it an argument distinct from any that it could obtain by exchanging.

Figure 7(a) shows the possible matches of exchange types within a channel. Figure 7(b) (contributed to the understanding of exchange functions by Dan Friedman, [Filman & Friedman 80]) shows the derivation of the three types. There must be both fully synchronized primitives ("synchronizing"), and also those which do not synchronize themselves ("free-running"). There must be exchanges which can match with their own kind and those that compete with their own kind. This makes four possibilities, except that a free-running type which exchanges with its own kind would be impossible, because it would require "matching" two simultaneous, instantaneous events.

5.3.2. Examples of fully synchronized interactions

In this section we will use exchange functions to specify the interactions between multiple transaction-processing terminals and a central database. "transact" in the terminal specification of 5.2 can be elaborated as follows:

```
transact[r] =
     receive-response[send-request[r]];

send-request:  REQUEST ---> FILLER;

send-request[r] = xm-requ[r];

receive-response:  FILLER ---> RESPONSE U ERROR-MESSAGE;
```

```
     receive-response[null] = x-resp['null'],
```

where "FILLER" is an intrinsic set whose only element is the constant "'null'". The database process successor function is specified as:

```
     database-cycle[d] =
          finalize-transaction[perform-transaction[(d,receive-request)]];

     receive-request:  ---> REQUEST;

     receive-request = x-requ['null'];

     perform-transaction:  DATABASE x REQUEST ---> DATABASE x RESPONSE;

     finalize-transaction:  DATABASE x RESPONSE ---> DATABASE;

     finalize-transaction[(d,r)] = proj-2-1[(d,send-response[r])];

     send-response:  RESPONSE ---> FILLER;

     send-response[r] = x-resp[r].
```

By renaming (redefining) the exchange functions, we are able to give them mnemonic names, and also to attach explicit types at the most meaningful place.

"send-request" in a terminal and "receive-request" in the database match with each other to transmit the request. Note that the type "xm"'s in the terminals compete for the type "x" in the database; if nothing but "x"'s were used, two evaluations of "send-request" in different terminals might match with each other! Since the "xm" and "x" are symmetric with respect to synchronization, either may have to wait for the other.

After the request is processed against the database, "finalize-transaction" disposes of the results. It is defined in terms of the intrinsic function "proj-2-1", which projects an ordered pair onto its first component, in this case the updated database. The second component is evaluated only for its side-effect of sending the response back; the "'null'" value it returns is thrown away.

"receive-response" could have been defined using type "xm", but an "x" is also correct, because precedence constraints enforced by the functional nesting of "send-request" inside "receive-response" ensure that at most one instance of "receive-response" will be in evaluation at any one time, namely that of the process whose request is now being processed. Thus matching on the channel "resp" is always unique.

"FILLER" and "´null´" are used as place-holders whenever syntactic rules dictate that there must be a set or value, but no semantics need be carried. "send-request", for instance, returns the value "´null´" because every function must have a value. And "receive-response" has "FILLER" as its domain simply because it is composed with "send-request", although the reason for the composition is sequencing rather than transfer of values. "receive-request" does not need a domain (because a function does not require one), but "x-requ" must have an argument—and so "´null´" is used.

5.3.3. Examples of free-running interactions

A "free-running" process is one whose only interactions occur via "xr", so that it will never wait to synchronize with another process. The prototypical free-running process is a real-time clock, which "ticks" once per process step, and could not fulfill its intended function if it had any synchronizing interactions. Such a process is specified:

```
(clock-cycle[0], . . . );
clock-cycle:  TIME ---> TIME;
TIME = INTEGER;
clock-cycle[t] = proj-2-1[(increment[t],offer-time[t])];
increment:  TIME ---> TIME;
offer-time:  TIME ---> FILLER U TIME;
offer-time[t] = xr-time[t].
```

Any process wishing to read the current time must evaluate:

```
    current-time:  ---> TIME;

    current-time = xm-time['null'].
```

Concurrent "xm-time"'s will compete to match with "xr-time", implying for this particular specification that no two readers will ever get the same clock value.

Another common type of free-running process is a digital simulation of a nondigital, unintelligent environment object. Here is the top-level specification of the processes representing the machines in the environment of a process-control system ([Zave & Yeh 81]):

```
    j#1..3

    < ; machine-j-cycle:  MACHINE-STATE ---> MACHINE-STATE;

        machine-j-cycle[m] =
             proj-2-1[(simulate-machine[(m,feedback-j-if-any)],
                       offer-machine-j-data[sense[m]]
                    )];

        feedback-j-if-any:  ---> FEEDBACK U FILLER;

        feedback-j-if-any = xr-j-back['null'];

        offer-machine-j-data:  SENSOR-DATA ---> FILLER U SENSOR-DATA;

        offer-machine-j-data[s] = xr-j-sens[s]

    >;

        simulate-machine:
             MACHINE-STATE x (FEEDBACK U FILLER) ---> MACHINE-STATE;

        sense:  MACHINE-STATE ---> SENSOR-DATA.
```

During each process step two things are done in parallel: (1) "simulate-machine" computes the next process state, which is an element of "MACHINE-STATE" encoding the machine's current status, and (2) the current output of sensors attached to the machine ("sense[m]") is offered to the control system via "xr-j-sens". If the control system is ready to accept the data from this machine cycle an exchange will take place; otherwise the data will be gone forever.

"simulate-machine" has two arguments: the current machine state, and the value returned by "feedback-j-if-any". This function is defined as "xr-j-back", an exchange function which interacts with several sites in the control system which provide controlling feedback to the jth machine. If some actuator is being activated at the moment "xr-j-back" is evaluated, an exchange takes place and a value in "FEEDBACK" is returned. Otherwise the argument "´null´" is returned, indicating that no actuators are being used.

Our final example of a free-running process is a producer-consumer buffer. Its process state is the current buffer contents, and its successor function is:

```
next-buffer:  BUFFER ---> BUFFER;

next-buffer[b] = give-to-consumer[get-from-producer[b]];

get-from-producer:  BUFFER ---> BUFFER;

get-from-producer[b] = /full[b]:  b,
                        ´true´ :  put-on-tail[(b,xr-prod[´null´])]
                       /;

give-to-consumer:  BUFFER ---> BUFFER;

give-to-consumer[b] =
     /empty[b]:  b,
      ´true´  :  put-on-head[(xr-cons[first[b]],rest[b])]
     /.
```

On each process step "get-from-producer" provides the opportunity to put one new element in the buffer (assuming it is not already full). If some producer has a pending "xm-prod[new-element]", "new-element" will be returned as the value of "xr-prod" and inserted. Otherwise "xr-prod" returns "´null´", which "put-on-tail" will simply ignore.

Likewise, on each process step "give-to-consumer" offers the element at the head of the buffer ("first[b]") to any process evaluating "xm-cons['null']". If such an evaluation is pending an exchange will take place and "xr-cons[first[b]]" will return "'null'", which "put-on-head" will ignore. Otherwise the unconsumed "first[b]" will be returned, and "put-on-head" will reinstate it.

The expected behavior of this process (at least under light loading) will be to cycle very fast, checking for interactions but not having any on most process steps. This shows that exchange functions are in some sense more primitive than synchronization mechanisms which enable a process to wait for any one of several events to occur. The payoff is a much simpler implementation for exchange functions, and the choice is in keeping with the PAISLey philosophy of simplicity and minimal semantics. It is also arguable that the above specification is as perspicuous as any, largely because of the benefits of applicative style.

5.3.4. Implementation

Exchange functions can be implemented straightforwardly, even on distributed networks (assuming a simple message-passing facility). In almost all cases the pattern of matches on a channel is one-to-one or many-to-one, the latter for resource competition. In this section we present an efficient distributed algorithm for implementing exchange matching in these cases.

Consider first an exchange channel with many "xm"'s and one "x" (or just two "x"'s, in which case one of them takes the role of the "xm" in this description), all residing at different nodes of a network (see Figure 8). When an "xm" is initiated, a message carrying its argument is sent to the node where the matching "x" resides. These messages are queued up in arrival order. When the "x" is initiated, if the queue is empty, it waits until it is not. When the queue is not empty, it removes the first entry as the "match", takes the value stored there as its own value, sends a termination message containing its argument to the matching "xm", and continues. Computation can continue at the "xm" as soon as the termination message (with its value) is received.

This implementation uses only two messages per match, and automatically prevents lockout with FIFO queueing. For channels with one "xr" and either one "x" or many "xm"'s, the queue is formed at the site of the "xr", and the only modification necessary is that if the "xr" is initiated when the queue of possible matches is empty, then it does not go into the wait state. For channels with many "xr"'s and one "x", matching is done at the site of the "x", but no queue forms. If an "xr" sends an initiating message but the "x" is not pending, a termination message with the original argument is sent back to the "xr" immediately.

In the rare cases where one party to all matches on a channel is <u>not</u> predetermined by the static specification structure, matching on a channel can be implemented by a "channel controller" situated anywhere in the network. The initiation of any exchange causes a message containing its type and argument to be sent to the controller. The controller queues and matches as appropriate, notifying an exchange function that it has been matched by sending it a termination message containing the value. This is less desirable than the previous strategy only because it requires four messages per match.

5.3.5. Further properties and justifications

Because exchange functions are only "pseudo-functions" and have side-effects, expressions containing them cannot be optimized to avoid evaluation of expressions whose <u>values</u> are not needed. The most common example of this is a successor function with the form "proj-2-1[(a,b)]", where expression "a" computes the next state and "b" interacts with other processes.

There is also a potential problem with distributing values obtained by interaction, but the formal parameter mechanism does this nicely. Suppose the effect of

```
/equal[(x-denom['null'],0)]:  'divide-check',
 'true':                      divide[(numerator,x-denom['null'])]
/
```

is wanted, where both usages of the value returned by an exchange are supposed to result from a single evaluation. This can be specified unambiguously by defining "quotient" as:

```
quotient[(n,d)] = /equal[(d,0)]:  'divide-check',
                   'true'      :  divide[(n,d)]
                  /,
```

and then using it in the invocation "quotient[(numerator,x-denom['null'])]".

Establishing the internal consistency of a specification with exchange functions requires some attention. The range of a user-chosen function defined as an exchange must agree with the domains of all those with which it can exchange. Furthermore, precedence constraints caused by nested evaluation structures can cause exchange deadlocks. But the channel of an exchange function has been made a constant attribute rather than an argument to it just so that exchange patterns would yield to static analysis, and simple arguments do establish deadlock-freedom in many common cases. For instance, the process hierarchy constructed in [Zave & Yeh 81] expresses the acyclic "dependency" structure of the interactions in the system; the argument that this prevents deadlock is a common one in the operating system literature (e.g. [Brinch Hansen 77]).

There are so many proposals for distributed interaction mechanisms current today that comparison and justification are essential. Most properly, exchange functions are motivated and justified by our goal of fitting processes and asynchronous interactions into an applicative framework, and in this role they are almost unique (see also [Milne & Milner 79]). Their generality is established by Figure 7(b) and by extensive experience with them, which indicates that the only kind of interaction they cannot specify is _unbounded_ broadcast.

Exchange functions can also be justified, however, on the same basis as procedure-based mechanisms, which fall into the two general categories of procedure-call mechanisms ([Brinch Hansen 78], [Hoare 74], [Ichbiah _et al._ 79]) and message-passing ([Rao 80]). Exchange functions are more primitive than procedure calls because they only specify interaction at one point in time rather than two (procedure call and return). They are thus more general and easier to implement, while the mutual synchronization of the communicating processes provides much of the structure and control usually associated with procedure-call mechanisms.

It is the mutual synchronization that most distinguishes exchange functions from message-passing mechanisms, where (usually) messages are automatically buffered, so that the sender transmits the message and continues, while the message is queued until the receiver is ready for it. The decision against this scheme is based on our concern with performance. Consider a set of terminals sending updates to a central database. With exchange functions a terminal cannot create new work for the system until the system has accepted its previous work. If a terminal could simply send an update message and continue, its speed could increase (unchecked by the ability of the system to handle the work), the queue at the database could grow to unbounded lengths, and no bounds on the performance of the system could ever be established.

At the same time, there is nothing wrong with _bounded_ buffering, but this can always be specified in PAISLey. Introducing bounds within an abstract, general-purpose interaction mechanism (such as "message passing up to some bound") would seem a most unfortunate mixture of specification and implementation.

Given that synchronization is going to be two-way, it costs very little in the implementation to preserve the possibility of two-way data transfer, although it is seldom used. It also keeps the number of primitives down by a factor of two, since otherwise each of the three exchange function types would have to come in a "sending data" and a "receiving data" version.

Of all the well-known interaction mechanisms, the most similar to exchange functions is Hoare's input/output primitives. In Hoare's language a pair of statements, "P?input" in process Q and "Q!output" in process P, will come together in the same mutually synchronized manner that two matching exchanges do. "output" is an expression whose value is assigned to the variable "input", assuming appropriate type correspondences. In addition to the relatively unimportant data asymmetry, Hoare's primitives seem to be different from exchange functions in three fundamental ways: (1) There is no way to specify real-time or free-running interactions. (2) There is no straightforward way to specify resource sharing, since all "matches" are one-to-one by process name. In Hoare's language a process representing a shared resource must have a separate command for each process with which it can communicate, and guard that command ([Dijkstra 75]) with an input command naming the appropriate process of the many. The guard (and statement) to be executed are chosen nondeterministically from the processes that are ready to communicate. These multiple statements seem distinctly clumsy compared to an "xm"/"x" exchange match. Furthermore, the full knowledge each process must have about the names of the processes with which it communicates makes modularity difficult to achieve. (3) Hoare's primitives belong in a procedural, rather than applicative, framework. The destination of a data transfer, for instance, is specified by an address.

5.4. Performance requirements

5.4.1. Definition of performance requirements

So far the only structure that has been needed for complete and formal specification of performance requirements is attachment of timing and reliability attributes to functions in the "functional" requirements specification. A timing attribute refers to the evaluation time of the function. It is a random variable, and any information about its distribution, such as lower or upper bounds, mean, or the distribution itself, may be given.[5]

A reliability attribute can only be attached to a function whose range is divided into two subsets (e.g. "---> SUCCESS-RESULT U FAILURE-RESULT"), the first for the values returned by successful evaluations and the second for values returned when the evaluation fails. The attribute itself is a discrete (binary) random variable whose two outcomes denote successful or failed evaluations, and any information about its distribution may be given.

When a primitive function fails it simply returns a value in its failure range. An exchange function fails in the same way, except that even when it fails it must still match another exchange—so that failures do not affect or complicate analysis of exchange patterns. Furthermore, if matching exchange functions both have reliability properties, they must succeed or fail together.

Failure of a nonprimitive function simply means that the function delivers a value in the second subset of its range. It is up to the function's definition to ensure that this happens with the specified frequency, since the function is evaluated according to its definition under all circumstances.

Reliability is a difficult and little-understood subject, but this definition of it has several appealing properties. It forces the specified system to have the primary characteristic of a reliable system, namely going into a well-defined and previously anticipated state when something fails. It makes reliability independent of timing and functionality, since a function evaluation must satisfy its timing requirements and deliver a value in its declared range regardless of whether it succeeds or fails. In fact, we have deemed this property so important that we have sacrificed some realism for it: only primitive functions can <u>really</u> fail (since nonprimitive ones are always evaluated according to their definitions). Much more knowledge of reliability is needed before we can be sure how successful this approach will be, but its formality and tractability are strong arguments in its favor.

These performance requirements can be simulated by the specification interpreter, and checked (in principle!) for internal consistency, just as the functional ones are. This means, for instance, that if "f[x]" is defined as "g[h[x]]", and there are upper bounds on the evaluation times of all three, then the upper bound on "f" must be strictly greater than (allowing time for invocation/argument transfer) the sum of the upper bounds on "g" and "h".

5.4.2. Examples of "synchronous closed-loop" performance requirements

An on-line database system can be called a "synchronous closed-loop" system--"closed-loop" because the entire feedback loop realized by the system is explicitly represented, and "synchronous" because the terminal process (on behalf of the cooperative person behind it) waits for responses, i.e. synchronizes itself with the system. For these systems the basic performance requirements are particularly easy to specify, and all are attached to the terminals. We will refer to the functional terminal specification in 5.2, and give performance requirements typical of airline reservation systems ([Knight 72]).

Distributions have been left as comments in the PAISLey syntax because we have not yet settled on a formal language for them. Timing requirements normally call for the maximum, minimum, mean, or constant value of the random variable, while reliability requirements normally specify a lower bound on the probability of success.

A response-time limit of 3 seconds is specified by:

```
transact:  ! ---> "maximum = 3 sec".
```

An average load of 200 transactions per second, assuming 10,000 terminals in the system, is specified by:

```
terminal-cycle:  ! ---> "mean = 50 sec",
```

which says that on the average a terminal demands a transaction (goes through a cycle) every 50 seconds. Finally, the requirement that at least 99 per cent of all transactions must be processed successfully is expressed as:

```
transact:  % ---> "prob{ 'success' } >= .99",
```

which, of course, can only be attached to "transact" because its range is divided into success ("RESPONSE") and failure ("ERROR-MESSAGE") subranges.

5.4.3. Examples of "asynchronous closed-loop" performance requirements

Process-control systems can be called "asynchronous closed-loop" systems--"asynchronous" because the machines, which are the sources and destinations of the feedback loops realized by these systems, are free-running. The systems must keep up with them without their cooperation. Performance requirements for these systems are more of a challenge, but the operational approach enables us to specify them straightforwardly. "Open-loop" specifications, in which not all of the feedback loop (ultimately, the purpose of any embedded system is to realize feedback loops) is included explicitly in the model, have performance requirements similar to these. An example of an open-loop specification would be a patient-monitoring system in which treatment of patients was not represented, only display of warning messages.

In [Zave & Yeh 81] a variety of timing requirements for a process-control system are given. These include: (1) the granularity of the discrete simulation of the machines in the environment, (2) a real-time limit on the fully automatic feedback loop realized by the system, (3) a real-time limit on the partially manual feedback loop realized by the system, specified as separate response requirements on the human operator and on the computer system, and (4) a derived performance requirement on the system's internal database component which will guarantee that other system components can meet their own performance requirements. Despite the subtleties involved, each one of these requirements is specified simply by attaching a timing constraint to the successor function of a single process!

5.4.4. "Real-world" properties

Time and reliability (the fact that sometimes digital components do not do what their definition says they will, for physical reasons forever beyond the reach of digital logic) are nondigital properties that incontrovertibly affect the digital domain. In [Zave 80b] many other such physical ("real-world") properties are mentioned, weight and distance, for example.

Why aren´t these formalized as performance requirements as well?

The answer is that, to the extent that we know them, the effects of these properties on the computational (digital) domain can be specified in terms of functions, timing, and reliability. Weight constraints, for instance, only affect how many functions can be realized. Even if we did attach weight attributes to components of a PAISLey specification, there is nothing that an interpreter could do with them. Therefore an informal comment is just as satisfactory.

Distance is a more interesting example because its effects on the computational domain are more varied. Distance increases the relative time for interprocess interactions, decreases component reliability, and increases the logical complexity of interfaces which must cope with these factors. Yet these three effects are directly expressable in terms of timing, reliability, and functional requirements, respectively.

Factors such as these can nevertheless have a profound effect on requirements. In an airline reservation system, for instance, it may be necessary to divide the response-time or transaction-reliability allowances into portions for the data-communication subsystem and portions for the database subsystem. Although such allocation is technically a design decision, two reasons for doing it during the requirements phase are: (1) to enable feasibility analyses of two very different technologies, and (2) to contract the work to different organizations.

These allocated requirements can be specified in PAISLey. We have constructed a requirements model in which time limits are given for, and failures can occur in, each of three stages: input transmission, transaction processing, and output transmission. Failure at any stage aborts subsequent stages and propagates an appropriate error message. This is the source of elements in the set "ERROR-MESSAGE" found in the range of "transact" in the terminal specification.

6. EVALUATION AGAINST GOALS FOR REQUIREMENTS SPECIFICATIONS

A thorough evaluation of PAISLey cannot be made until an interpreter has been implemented and specifications of large systems have been written. In the meantime, we present the reasons why we believe our goals for requirements specifications will be met.

It was stated in 1.2 that, in order to constrain the target system, a requirements specification should be precise, unambiguous, internally consistent, complete, and minimal. The formal nature of a PAISLey specification leaves no doubt as to its precision and lack of ambiguity. If it (1) is syntactically correct, (2) has all its domain declarations in agreement with all its function applications, (3) has all its range declarations in agreement with its function definitions, and (4) can be shown to be free of exchange deadlock, then it is internally consistent. The arguments in 4.1 suggest that PAISLey does not obstruct the writing of minimal specifications.

The issue of completeness deserves special attention, because the worst failing a requirements specification language can have is to be unable to express what the requirements analyst wants to say. This is the problem that makes analysts revert to English! PAISLey has been used to specify requirements for a wide variety of embedded systems (small examples), and never yet found wanting. In addition to the examples used or referred to here, it was also used to specify (in some 33 pages, see [Zave 78]) a distributed design for an innovative interactive numerical system that was actually implemented directly from the specification ([Zave & Rheinboldt 79], [Zave & Cole 81]). Furthermore, the prompt feedback provided by executable requirements should help to protect against omissions.

Of course, this refers only to properties relevant to the computational domain--it says nothing about constraints on the development process itself, such as deadlines, cost limits, methodological standards, and routine maintenance procedures. PAISLey does not address these, nor does it offer any particular help in posing alternate or prioritized requirements ([Yeh et al. 80]). And the need to supplement formal requirements with diagrams, comments, and other informal avenues of human communication will never disappear.

A requirements specification must be formally manipulable or testable, so that it can be determined whether the final product meets its requirements. Any property of a PAISLey specification relating to the behavior of the system in a given situation is clearly testable, because the behavior of the executable specification can be compared directly to the behavior of the target system. PAISLey specifications should also be suitable for verification, because they are based on a few simple and formal concepts, but this aspect has not yet been explored.

Finally, requirements specifications should be understandable and modifiable. Although these qualities are vague and subjective, there is reason to believe that they can both be achieved by attaining two other qualities which are at least identifiable, if still subjective. These two qualities are: (1) providing modeling capabilities which are abstract, intuitive, and close to human perceptions of the environment to be modeled, and (2) providing means by which complexity can be decomposed. The latter is necessary because a complex specification must be understood (and changed) one small piece at a time. The former is important for understanding because it allows people to think in familiar, problem-oriented terms. It may also be important for modifiability, as explained in [Goldman & Wile 80]. They argue that, since our current best definition of modifiability is that a small change *in the environment* causes *a* correspondingly small change in the specification, the property is most possessed by those specifications which are the most direct translations of the natural environment into formalism.

We contend that PAISLey models can be intuitive and close to the natural way people think about embedded systems. The objects in the system's environment can be modeled directly as autonomous parallel processes, using a language with the power to specify simulations of great verisimilitude. The internal process structure of the proposed system model is largely determined by the environment structure and user-oriented system "functions" or capabilities (see 7.2).

PAISLey also enables nontrivial decomposition of complexity. The division of a specification into processes is especially useful, for instance, because it decomposes both static and dynamic properties, and because it seems to correlate with divisions meaningful to users. The partitioning even extends to execution of specifications, because any subset of processes can be executed in isolation, simply by leaving all interactions with missing processes as unelaborated primitives in a form such as "receive-message: ---> MESSAGE". The interpreter will evaluate this "interaction site" by choosing some message at random. This capability was used in [Zave & Yeh 81] to develop a specification in five versions, each independently executable, and each obtained from the last by adding new processes/functions in an "outside-in" sequence. *Decomposition of complexity* is discussed further in [Zave 80a].

7. PLANS FOR FUTURE RESEARCH

7.1. Execution of Specifications

We are currently planning the implementation of a system for executing specifications. Specifications will be checked for consistency and compiled into fully parallel runtime process-and-interaction structures. These structures will then be interpreted under interactive control. The most important questions to be answered are: What is it like to test a specification? What kind of control over the course of execution does the analyst need? What display, report, and trace facilities can be used to produce an intelligible outcome?

7.2. Methodology

This work on requirements specification, which has been pursued so far with small examples, must be extended in the directions of requirements analysis, and "scaling-up" to large systems. In other words, we need to pursue the implications of process-based specifications for a requirements methodology.

The most obvious problem is that process-based descriptions are not intrinsically hierarchical. It is not clear, however, that top-level system requirements are hierarchical either. Preliminary studies indicate that a complex system carries out several parallel functions for its environment, none of which is subservient to any other. ("Top-level" is important here because once the process structure of a PAISLey specification is determined, further decisions are recorded by elaboration of successor functions--which is completely hierarchical.)

This observation has suggested an alternate methodological approach, in which parallel functions are identified and translated into PAISLey processes or hierarchically arranged groups of them (similar to "subsystems" in DDN, see [Riddle et al. 78]). Preliminary work with three process-control examples has even revealed a set of rules which can be used to identify user-oriented functions of these systems; functions defined according to these rules have a one-to-one correspondence with processes in the subsequent PAISLey specification.

Complexity is then decomposed through incremental development: the specification is written and tested one function/process at a time. The language semantics already support this mode of use, as explained in Section 6, and our investigations of the interpreter's human interface are also being carried out with the incremental mode in mind.

These ideas are still highly speculative, but they do establish that hierarchical abstraction is not the only approach to a practical requirements methodology. At the highest levels of system description, "flat" structure may not be incompatible with good structure.

7.3. Design

It has been pointed out that PAISLey is capable of specifying the results of design decisions. A logical extension of this is to investigate its properties as a design specification language. The benefits are potentially great, because a uniform language for requirements and design should make possible substantial improvements in the traceability and automatability of design. It might also lead to a better theoretical understanding of design decisions as resource/performance trade-offs.

ACKNOWLEDGMENTS

Many people have contributed to the work presented here. My thanks especially to Bob Fitzwater, with whom the foundations for PAISLey were laid, to Steve Smoliar, Alex Conn, and Roland Mittermeir, for stimulating discussions on requirements, to Priscilla Fowler, for the chance to try these ideas on a class at Bell Labs, to George Cole, for diligent and able assistance, to Dick Hamlet, who would read this stuff before anyone else would (or could), to the referees, and to Raymond Yeh, for support, encouragement, ideas, and good advice.

REFERENCES

[Air Force 65]
U.S. Air Force, "Air Force Weapons Effectiveness Testing (AFWET) Instrumentation System", R&D Exhibit No. PGVE 64-40, Air Proving Ground Center, Eglin Air Force Base, Florida, 1965.

[Alford 77]
Mack W. Alford, "A Requirements Engineering Methodology for Real-Time Processing Requirements", IEEE Trans. Software Engr. SE-3, January 1977, pp. 60-69.

[Backus 78]
John Backus, "Can Programming be Liberated from the von Neumann Style? A Functional Style and its Algebra of Programs", Comm. ACM 21, August 1978, pp. 613-641.

[Balzer & Goldman 79]
Robert Balzer and Neil Goldman, "Principles of Good Software Specification and Their Implications for Specification Language", Proc. Specifications of Reliable Software Conf., Cambridge, Mass., April 1979, pp. 58-67.

[Belady & Lehman 79]
L.A. Belady and M.M. Lehman, "The Characteristics of Large Systems", Research Directions in Software Technology, Peter Wegner, ed., M.I.T. Press, Cambridge, Mass., 1979, pp. 106-138.

[Bell et al. 77]
Thomas E. Bell, David C. Bixler, and Margaret E. Dyer, "An Extendable

Appraoch to Computer-Aided Software Requirements Engineering", IEEE Trans. Software Engr. SE-3, January 1977, pp. 49-60.

[Bell & Thayer 76]
T.E. Bell and T.A. Thayer, "Software Requirements: Are They Really a Problem?", Proc. 2nd Intl. Conf. on Software Engineering, San Francisco, Cal., October 1976, pp. 61-68.

[Boehm 76]
Barry W. Boehm, "Software Engineering", IEEE Trans. Computers C-25, December 1976, pp. 1226-1241.

[Brinch Hansen 77]
Per Brinch Hansen, The Architecture of Concurrent Programs, Prentice-Hall, Inc., 1977.

[Brinch Hansen 78]
Per Brinch Hansen, "Distributed Processes: A Concurrent Programming Concept", Comm. ACM 21, November 1978, pp. 934-941.

[Conn 80]
Alex Paul Conn, "Maintenance: A Key Element in Computer Requirements Definition", Proc. COMPSAC ´80, Chicago, Ill., October 1980, pp. 401-406.

[Davis & Rauscher 79]
Alan M. Davis and Tomlinson G. Rauscher, "Formal Techniques and Automatic Processing to Ensure Correctness in Requirements Specifications", Proc. Specifications of Reliable Software Conf., Cambridge, Mass., April 1979, pp. 15-35.

[Davis & Vick 77]
Carl G. Davis and Charles R. Vick, "The Software Development System", IEEE Trans. Software Engr. SE-3, January 1977, pp. 69-84.

[Dijkstra 75]
E.W. Dijkstra, "Guarded Commands, Nondeterminacy, and Formal Derivation of Programs", Comm. ACM 18, August 1975, pp. 453-457.

[Fisher 78]
David A. Fisher, "DoD´s Common Programming Language Effort", Computer 11, March 1978, pp. 24-33.

[Filman & Friedman 80]
Robert E. Filman and Daniel P. Friedman, Languages and Models for Distributed Computing, to appear.

[Fitzwater & Zave 77]
D.R. Fitzwater and Pamela Zave, "The Use of Formal Asynchronous Process Specifications in a System Development Process", Proc. 6th Texas Conf. on Computing Systems, Austin, Texas, November 1977, pp. 2B-21 - 2B-30.

[Friedman & Wise 77]
Daniel P. Friedman and David S. Wise, "Aspects of Applicative Programming for File Systems", Proc. ACM Conf. on Language Design for Reliable

Software, Raleigh, N. Car., March 1977, pp. 41-55.

[Friedman & Wise 78a]
Daniel P. Friedman and David S. Wise, "Aspects of Applicative Programming for Parallel Processing", IEEE Trans. Computers C-27, April 1978, pp. 289-296.

[Friedman & Wise 78b]
Daniel P. Friedman & David S. Wise, "Unbounded Computational Structures", Software--Practice and Experience 8, July-August 1978, pp. 407-416.

[Friedman & Wise 79]
Daniel P. Friedman and David S. Wise, "An Approach to Fair Applicative Multiprogramming", Semantics of Concurrent Computation (G. Kahn, ed.), Lecture Notes in Computer Science 70, Springer-Verlag, Berlin, 1979, pp. 203-226.

[Friedman & Wise 80]
Daniel P. Friedman and David S. Wise, "An Indeterminate Constructor for Applicative Programming", Proc. 7th Annual ACM Symp. on Principles of Programming Languages, Las Vegas, Nev., January 1980, pp. 245-250.

[Goldman & Wile 80]
Neil M. Goldman and David S. Wile, "A Relational Data Base Foundation for Process Specifications", Entity-Relationship Approach to Systems Analysis and Design, P.P. Chen, ed., North-Holland, 1980.

[Heninger 79]
Kathryn L. Heninger, "Specifying Software Requirements for Complex Systems: New Techniques and Their Application", Proc. Specifications of Reliable Software Conf., Cambridge, Mass., April 1979, pp. 1-14.

[Hoare 74]
C.A.R. Hoare, "Monitors: An Operating System Structuring Concept", Comm. ACM 17, October 1974, pp. 549-557.

[Hoare 78]
C.A.R. Hoare, "Communicating Sequential Processes", Comm. ACM 21, August 1978, pp. 666-677.

[Horning & Randell 73]
J.J. Horning and B. Randell, "Process Structuring", Computing Surveys 5, March 1973, pp. 5-30.

[Ichbiah et al. 79]
J.D. Ichbiah et al., "Rationale for the Design of the Ada Programming Language", SIGPLAN Notices 14, June 1979, Part B.

[Ingalls 78]
Daniel H.H. Ingalls, "The Smalltalk-76 Programming System Design and Implementation", Proc. 5th Annual ACM Symp. on Principles of Programming Languages, Tucson, Ariz., January 1978, pp. 9-16.

[Iverson 80]

Kenneth E. Iversion, "Notation as a Tool of Thought", Comm. ACM 23, August 1980, pp. 444-465.

[Knight 72]
John R. Knight, "A Case Study: Airlines Reservations Systems", Proc. of the IEEE 60, November 1972, pp. 1423-1431.

[Lehman 80]
Meir M. Lehman, "Programs, Life Cycles, and Laws of Software Evolution", Proc. of the IEEE 68, September 1980, pp. 1060-1076.

[Mao & Yeh 80]
William T. Mao and Raymond T. Yeh, "Communication Port: A Language Concept for Concurrent Programming", IEEE Trans. Software Engr. SE-6, March 1980, pp. 194-204.

[Milne & Milner 79]
George Milne and Robin Milner, "Concurrent Processes and Their Syntax", Jour. ACM 26, April 1979, pp. 302-321.

[Mittermeir 80]
Roland T. Mittermeir, "Semantic Nets for Modeling the Requirements of Evolvable Systems--An Example", Institut fuer Digitale Anlagen, Technische Universitaet Wien, Vienna, Austria, May 1980.

[Rao 80]
Ram Rao, "Design and Evaluation of Distributed Communication Primitives", Univ. of Wash. Computer Science 80-04-01, Seattle, Wash., April 1980.

[Riddle et al. 78]
William E. Riddle et al., "Behavior Modeling During Software Design", IEEE Trans. Software Engr. SE-4, July 1978, pp. 283-292.

[Ross 77]
Douglas T. Ross, "Structured Analysis (SA): A Language for Communicating Ideas", IEEE Trans. Software Engr. SE-3, January 1977, pp. 16-34.

[Ross & Schoman 77]
Douglas T. Ross and Kenneth R. Schoman, "Structured Analysis for Requirements Definition", IEEE Trans. Software Engr. SE-3, January 1977, pp. 6-15.

[Roussopoulos 79]
Nicholas Roussopoulos, "CSDL: A Conceptual Schema Definition Language for the Design of Data Base Applications", IEEE Trans. Software Engr. SE-5, September 1979, pp. 481-496.

[Smith & Smith 79]
John Miles Smith and Diane C.P. Smith, "A Data Base Approach to Software Specification", Proc. Software Development Tools Workshop, Pingree Park, Colo., May 1979, (Springer-Verlag, W.E. Riddle and R.E. Fairley, eds., 1980), pp. 176-200.

[Smoliar 79]

Stephen W. Smoliar, "Using Applicative Techniques to Design Distributed Systems", Proc. Specifications of Reliable Software Conf., Cambridge, Mass., April 1979, pp. 150-161.

[Smoliar 80]
Stephen W. Smoliar, "Applicative and Functional Programming", Software Engineering Handbook, C.V. Ramamoorthy and C.R. Vick, eds., Prentice-Hall, Inc., to appear.

[Teichroew & Hershey 77]
Daniel Teichroew and Ernest A. Hershey III, "PSL/PSA: A Computer-Aided Technique for Structured Documentation and Analysis of Information Processing Systems", IEEE Trans. Software Engr. SE-3, January 1977, pp. 41-48.

[Yeh et al. 79a]
Raymond T. Yeh et al., "Software Requirement Engineering--A Perspective", Univ. of Texas Computer Science SDBEG-7, Austin, Texas, March 1979.

[Yeh et al. 79b]
Raymond T. Yeh, Nick Roussopoulos, and Philip Chang, "Systematic Derivation of Software Requirements Through Structured Analysis", Univ. of Texas Computer Science SDBEG-15, Austin, Texas, 1979.

[Yeh et al. 80]
Raymond T. Yeh et al., "Software Requirements: A Report on the State of the Art", Univ. of Maryland Computer Science TR-949, College Park, Maryland, October 1980 (to appear as "Software Requirements: New Directions and Perspectives" in Software Engineering Handbook, C.V. Ramamoorthy and C.R. Vick, eds., Prentice-Hall, Inc.).

[Zave 78]
Pamela Zave, "The Formal Specification of an Adaptive, Parallel Finite-Element System", Univ. of Maryland TR-715, College Park, Maryland, December 1978.

[Zave 80a]
Pamela Zave, "The Operational Approach to Requirements Specification for Embedded Systems", Univ. of Maryland TR-976, College Park, Maryland, December 1980.

[Zave 80b]
Pamela Zave, "'Real-World' Properties in the Requirements for Embedded Systems", Proc. 19th Annual Wash., D.C. ACM Tech. Symp., Gaithersburg, Md., June 1980, pp. 21-26.

[Zave 81]
Pamela Zave, "Extending Applicative Specification Techniques to Embedded Systems", submitted for publication, 1981.

[Zave & Cole 81]
Pamela Zave and George E. Cole, Jr., "A Quantitative Evaluation of the Feasibility of, and Suitable Hardware Architectures for, an Adaptive, Parallel Finite-Element System", submitted for publication, 1981.

[Zave & Rheinboldt 79]
Pamela Zave and Werner C. Rheinboldt, "Design of an Adaptive, Parallel Finite-Element System", ACM Trans. Math. Software 5, March 1979, pp. 1-17.

[Zave & Yeh 81]
Pamela Zave and Raymond T. Yeh, "Executable Requirements for Embedded Systems", Proc. 5th Intl. Conf. on Software Engr., San Diego, Cal., March 1981, pp. 295-304.

APPENDIX: A GRAMMAR FOR PAISLEY

This grammar is LALR, and is written in BNF with nonterminals underlined.

Comments are transparent, and can therefore appear anywhere. Blanks are also transparent, except inside an ascii-string.

```
comment ::= "ascii-string"

------------------------------------------------------------------------

spec ::= spec ; statement |

         statement |

         spec ; index-head < ; spec > |

         index-head < ; spec >

index-head ::= lower-string # integer .. integer |

               upper-string # integer .. integer |

               # integer .. integer

statement ::= system-decl |

              func-decl |

              set-defn |

              func-defn

------------------------------------------------------------------------

system-decl ::= ( process-list )

process-list ::= process-list , process |

                 process |
                process-list , index-head < , process-list > |
                index-head < , process-list >
process ::= func-name [ func-exp ]
--------------------------------------------------------------------------
func-decl ::= func-name : func-property
func-property ::= domain-range |
                  timing-attribute |
                  reliability-attribute
domain-range ::= set-exp ---> set-exp |
                 ---> set-exp
timing-attribute ::= ! ---> comment
reliability-attribute ::= % ---> comment
--------------------------------------------------------------------------
set-defn ::= set-name = set-exp
set-exp ::= set-exp U set-term |
            set-term |
            set-exp U index-head < U set-exp > |
            index-head < U set-exp >
set-term ::= set-term x set-item |
             set-item |
             set-term x index-head < x set-term > |
             index-head < x set-term >
set-item ::= set-name |
             ( set-exp ) |
             { const-list }
set-name ::= upper-string - set-name-string |
             upper-string
```

```
set-name-string ::= set-name-string - set-syll |
                    set-syll

set-syll ::= upper-string |
             integer

const-list ::= const-list , const-name |
               const-name

const-name ::= 'ascii-string' |
               integer |
               real-number
```

```
func-defn ::= func-name = func-exp |
              func-name formal-params = func-exp

formal-params ::= [ param-list ]

param-list ::= param-list , func-name |
               func-name |
               param-list , ( param-list ) |
               ( param-list )

func-exp ::= func-name |
             const-name |
             func-appl |
             ( func-list ) |
             / pred-pair-list , 'true' : func-exp /

func-appl ::= func-name [ func-exp ] |
              index-head < func-name > [ func-exp ]

func-list ::= func-list , func-exp |
              func-exp |
              func-list , index-head < , func-list > |
                 index-head < , func-list >

pred-pair-list ::= pred-pair-list , pred-pair |

                   pred-pair |

                   pred-pair-list , index-head < , pred-pair-list > |

                   index-head < , pred-pair-list >

pred-pair ::= func-exp : func-exp

func-name ::= lower-string |

              lower-string - func-name-string

func-name-string ::= func-syll - func-name-string |

                     func-syll

func-syll ::= lower-string |

              integer
```

Primitives of the grammar.

ascii-string ::= any string of ASCII characters

upper-string ::= any string of upper-case alphabetical characters (note that "U" is also an operator, and should not be generated as a set-name)

lower-string ::= any string of lower-case alphabetical characters (note that "x" is also an operator, and should not be generated as a func-name)

integer ::= any string of numerals

real-number ::= any string of numerals with a single embedded period

---

Intrinsic sets.

FILLER = { ´null´ }

BOOLEAN = { ´true´, ´false´ }

INTEGER = the set of all integers representable on the host machine

REAL = the set of all real numbers representable on the host machine

STRING = the set of all string constants with length less than or equal to some bound

Typeless intrinsic functions.

x-<u>lower-string</u> | xm-<u>lower-string</u> | xr-<u>lower-string</u>

proj-<u>integer</u>-<u>integer</u>

equal

Typed intrinsic functions.

sum: INTEGER x INTEGER --> INTEGER

difference: INTEGER x INTEGER --> INTEGER

product: INTEGER x INTEGER --> INTEGER

quotient: INTEGER x INTEGER --> INTEGER

remainder: INTEGER x INTEGER --> INTEGER

greater-than: INTEGER x INTEGER --> BOOLEAN

less-than: INTEGER x INTEGER --> BOOLEAN

greater-than-or-equal: INTEGER x INTEGER --> BOOLEAN

less-than-or-equal: INTEGER x INTEGER --> BOOLEAN

FOOTNOTES

[Affiliation of Author] This research was supported in part by the Air Force Office of Scientific Research under Contract F49620-80-C-0001 at the University of Maryland.

The author is with Bell Laboratories, Murray Hill, New Jersey 07974.

[1] Thus "embedded" is almost synonymous with "real-time", but we prefer the newer term because it does not exclude performance requirements dealing with reliability.

[2] Throughout this paper mappings will be called "functions", despite the fact that mappings named in specifications are often relations. The reason is that "function" gives a more accurate impression: the intention is always to produce a unique value when the mapping is invoked in the eventual target system, even though that value cannot always be determined by a known functional expression.

[3] In most cases a practical requirements specification, even if executable, will be far more abstract (less detailed) than the system it specifies. Other differences between executable specifications and implementations lie in their performance, resource, and accessibility properties. A prototype falls somewhere between an executable specification an an implementation, because it must get some exposure in the field, and this requires performance, resource, and accessibility properties which are adequate at some minimal level.

[4] The matching of index syllables does not discriminate between upper

and lower case letters, so that an index syllable has scope over set and function names alike.

[5] More generally, the sequence of evaluations of the function over the lifetime of the system could be associated with a stochastic process, so that the time of each evaluation would be a separate random variable, but let us hope such generality will never be needed.

CAPTIONS FOR FIGURES

Figure 1. A requirements-level system classification.

Figure 2. Partial model of a patient-monitoring system.

Figure 3. Processes in action.

Figure 4. A data-access graph for filling orders from an inventory.

Figure 5. A stimulus-response path (part of a patient-monitoring system) specified in RSL (from [Alford 77]).

Figure 6. The conceptual model underlying both process-oriented and data-oriented system specifications.

Figure 7. The three types of exchange function.

Figure 8. Distributed implementation of exchange matching.

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. ~~REPORT NUMBER~~ AFOSR-TR- 81 - 0662	2. GOVT ACCESSION NO. AD-A105 459	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) AN OPERATIONAL APPROACH TO REQUIREMENTS SPECIFICATION FOR EMBEDDED SYSTEMS.		5. TYPE OF REPORT & PERIOD COVERED TECHNICAL
		6. ~~PERFORMING ORG. REPORT~~ NUMBER
7. AUTHOR(s) Pamela Zave		8. CONTRACT OR GRANT NUMBER(s) F49620-80-C-0001
9. PERFORMING ORGANIZATION NAME AND ADDRESS Department of Computer Science University of Maryland College Park MD 20742		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS PE61102F, 2304/A2
11. CONTROLLING OFFICE NAME AND ADDRESS Air Force Office of Scientific Research/NM Bolling AFB DC 20332		12. REPORT DATE AUGUST 1981
		13. ~~NUMBER OF PAGES~~ 66
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)		15. SECURITY CLASS. (of this report) UNCLASSIFIED
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Requirements analysis and specification, embedded (real-time) systems, system design and specification, simulation models, distributed processing, applicative programming.

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

The approach to requirements specification for embedded systems described in this paper is called 'operational' because a requirements specification is an executable model of the proposed system interacting with its environment. The approach is embodied by the language PAISLey, which is motivated and defined herein. Embedded systems are characterized by asynchronous parallelism, even at the requirements level; PAISLey specifications are constructed of interacting processes so that this can be represented directly. Embedded (CONT)

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

ITEM #20, CONT:

systems are also characterized by urgent performance requirements, and PAISLey offers a formal, but intuitive, treatment of performance.

DAT
FILM