THE FORMAL DESIGN AND ANALYSIS OF DISTRIBUTED DATA-PROCESSING SYSTEMS

by

D. R. Fitzwater

Computer Sciences Technical Report #295

March 1977

THE FORMAL DESIGN AND ANALYSIS OF DISTRIBUTED DATA-PROCESSING SYSTEMS *

by

D. R. Fitzwater

ABSTRACT

The research proposal is to support the development of the "science" behind software engineering in order to ensure required system properties, to compare current software engineering techniques, to develop specification for new design and analysis tools, and to demonstrate the practicality of the "science".

A hierarchical design schema will be developed within which formal representations and analyses can be defined and the required solutions can be found. Since "worst-case" problems are generally impossible to solve, sufficient design laws or constraints will be developed to ensure solvability of the critical problems.

*Sponsored by the Ballistic Missile Defense Systems Command, Contracts Office, BMDSC-CRS, P.O. Box 1500, Huntsville, AL 35807, under Contract No. DASG60-76-C-0080.

Report CSTR 295

THE FORMAL DESIGN AND ANALYSIS OF DISTRIBUTED DATA-PROCESSING SYSTEMS

D. R. Fitzwater
University of Wisconsin
Computer Sciences Dept.
1210 W. Dayton St.
Madison, Wisc. 53706

March 1977

Final Report for Period 1 July 1976 - 15 March 1977

Prepared for
BALLISTIC MISSILE DEFENSE SYSTEMS COMMAND
Ballistic Missile Defense Advanced Technology Center
Contracts Office, BMDSC-CRS
P. O. Box 1500
Huntsville, Alabama 35807

DIR ONR BRD
Room 286
536 South Clark Street
Chicago, Illinois 60605

TABLE OF CONTENTS

THE FORMAL DESIGN AND ANALYSIS OF DISTRIBUTED DATA PROCESSING SYSTEMS

1. INTRODUCTION

1.1 Background

This final report presents a summary of the current status of work under Contract DASG6Ø-76-C-ØØ8Ø with the Department of Defense, Army, Huntsville, Alabama. The intent of this report is to document the results of the review of the current state-of-the-art critical problems, development of a "top-down" approach, and proposals for more detailed solutions.

All of the results here must be considered as preliminary and subject to change and elaboration as this work proceeds. The problems are complex and certainty of results comes (if then) only when the study is completed with respect to a set of properties.

1.1.1 Problem Area

Experience has shown that the specification, design, implementation, and development of complex real-time weapons systems, such as ballistic missile defense systems, are very expensive, difficult to test adequately, slow to develop and deploy, and difficult to adapt to changing requirements

Sponsored by The United States Army under Contract No. DASG6Ø-76-C-ØØ8Ø

[DaV77]. The introduction of distributed data processing concepts potentially complicates these problems even more. No development of science or design and testing tools will make such development automatic or simple. The engineering decisions will remain complex and dependent on experience and analysis. As in all other engineering fields, the development of a science of design, however incomplete, can be very valuable in guiding engineering decisions, analyzing consequences, providing design laws sufficient to guarantee some desirable system invariancies, and in avoiding "worst case" type designs, thus making possible more powerful analysis tools.

1.1.1.1 Critical Issues

There are many critical issues in the process of developing and deploying a major system. We cannot hope to address directly most of them. Rather, we must address some of them in a way that minimally constrains potential solutions to the others. If we can't solve all of the problems (and we can't), we mustn't prevent others from doing their best on unsolved problems while exploiting our results on those problems we have attacked. We will look first at the problems of making and testing engineering decisions with the goal of making the development process more manageable and engineering decisions more testable at

[DaV77] Davis, Carl G., and Vick, Charles R. "The Software Development System," IEEE Transactions on Software Engineering, January 1977.

earlier stages of development. We believe that a relatively minor investment in better models, design laws, and testing tools will have a large payoff in both the resources involved in development, in speeding up the deployment, and in improving the adaptability of the resulting system [DrK76]. The final validation of this belief will rest on use of the processes developed in this work.

We will first give an informal characterization of a development process. A formal specification of a particular set of desirable development processes is one of the intermediate goals of this work. We will start with a very general concept and gradually develop it into a formal and practical scheme for system development.

If the development is to be formalized, the current state of the development must be well defined at the beginning and at least at the end. The passage from one well-defined state to the next will be called a step of the process. Most useful development processes will proceed via many successive intermediate well-defined states. Of course there may not exist any algorithm for carrying out a given process step, and most development processes are not guaranteed to succeed, given arbitrary originating requirements. Thus the development process is similar to ordinary discrete processes, except that its steps may not

[DrK76] Dreyfus, J.M., and Karacsony, P.J. "The Preliminary Design as a Key to Successful Software Development," Proceedings of the 2nd International Conference of Software Development, San Francisco, California (October 1976), p. 206.

be effective (i.e., there is no automatic way to carry out a given step), as is shown in Figure 1. Development processes may be decomposed into independent or interacting processes, just as can discrete processes. Indeed the most general model of development processes is just that of digital processes. The notions of "well-defined state" and of "interaction" must be formalized.

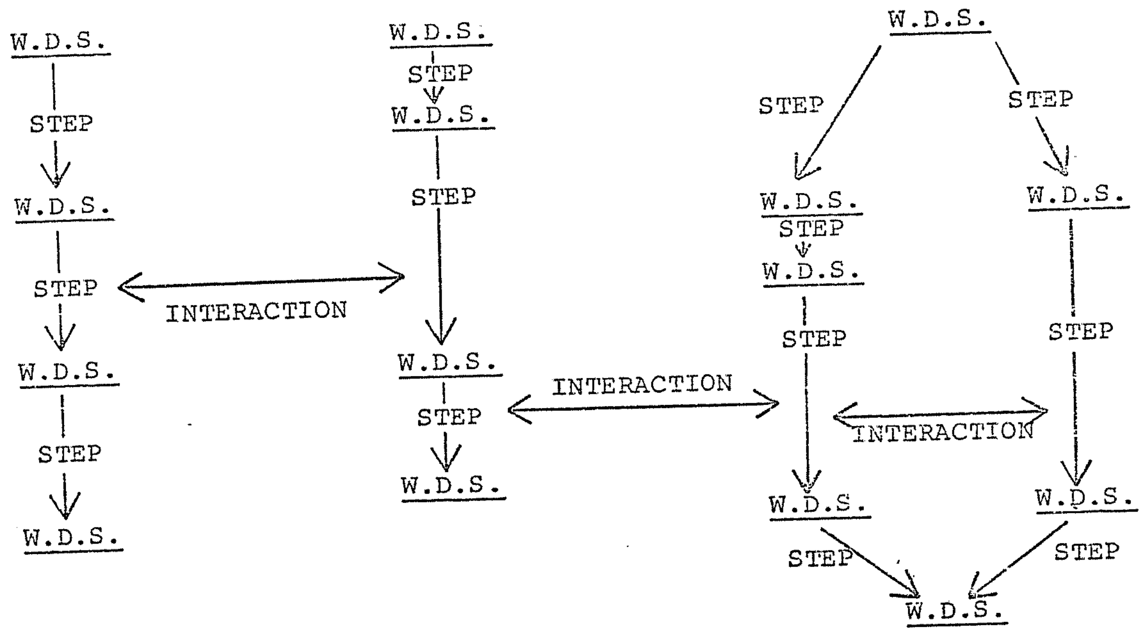


Figure 1. An interacting set of digital processes with interacting and non-interacting steps. The process is effective if each step is effective. W.D.S. represents "well-defined state."

Of course many digital processes do not model a desirable (or even feasible) development process; we must still sort through our models to find those with desirable properties.

We can identify at least the following processes involved in a system development process:

- Requirements (of the designed system)
- Design (of the specification of the system)
- Implementation (of the specified system)
- Evolution (changes in requirements or implementation design during operation)
- Operation (of the implemented system).

Each process has its own unique requirements for kinds of engineering decisions, of analysis, and of testability. However, all of these processes have much in common, and a general model with properties useful to all of them can be developed. At that high level of abstraction, all such results can be used for any of the above processes. In developing such an abstraction, we must find requirements for all such processes. Finding these requirements becomes a "meta-process" itself.

We will address the relevant critical issues arising from the processes above in the order given. Thus, this report will be most concerned with the top two processes and the "meta-process".

1.1.1.2 The Requirements Process

The requirements process starts with some (possibly incomplete, vague, and informal) originating requirements for a system that approximates the desired system, and finishes when the modified and elaborated requirements have been encoded (in a form suitable for the subsequent design process) and tested (to the satisfaction of the system engineers and the "customer"). The steps in the requirements process are of three types: the elaboration of requirements for an approximating system, the modification of requirements to those for a better approximation of the desired system, and the decomposition of the overall desired system into more manageable subsystems. There are critical issues involved with both the starting point and the ending point as well as for each type of step in the process.

1.1.1.2.1 At least the following critical issues are involved in the starting and ending points of the requirements process:

- What should be required?
  - e.g., Behavior in real world being affected by system,
    - Behavior at interface of real world and system
    - Functional structure of system
- What attributes should be constrained?
  - e.g., Which are bound by requirements?
    - Which are strongly coupled (engineering decisions can be made independently within a parametric range)?
- How should requirements be encoded?
  - e.g., To enable the testing of the requirements
    - To enable the design of the required system
    - To maximize applicability of tools and analysis

To check for consistency, completeness, etc.

To ensure that designed system is testable with respect to requirements.

Under worst case conditions, none of the above questions have satisfactory answers, so we must find design laws for requirements that make satisfactory answers possible.

1.1.1.2.2 At least the following are involved in a decomposition step:

- Decompose into what parts?

  e.g., Abstractions
  Sub-systems
  Levels

- How can requirements be allocated and tested?

  e.g., Testable in part
  Testable only on collection of parts
  Coupling between engineering decisions between parts

- How can parts be integrated?

  e.g., At completion of which process (requirements, design, implementation, etc.)
  At which stage of elaboration
  With minimal testing during integration

The decomposition must consider the subsequent integration and must make integration possible while maximizing the probability that requirements will be met.

1.1.1.2.3 At least the following are involved in a "better approximation" step:

- How good an approximation is it? (It should satisfy customer.)

  e.g., How to test behavior of required system

  How to compare with desired behaviors

  How to compare two approximating systems

- How can deficiences of approximation be corrected?

  e.g., Given analysis data, what are deficiencies?

  Should we change requirements or the desired system?

  What can we change to correct deficiency?

  What is impact of change on other requirements?

- How can the impact of change be determined?

  e.g., Either desired system or approximating system may change

  What other requirements are impacted?

  Which need to be changed?

  Which need to be retested?

  What is the traceability of requirements and interactions?

Since the question of "...better approximation to what?" must remain formally unanswered (or we would just change our starting point to an earlier form of requirement), we can not hope to use formal correctness proof techniques on originating requirements. Consequently we can only maximize the analysis information available to the customer and system engineers.

1.1.1.2.4 At least the following are involved in an elaboration of requirements step:

- Which process is constrained by the requirement?

  e.g., Requirements
  Design
  Implementation
  Etc.

- What type of testing is required?

  e.g., Stochastic
  Analytical
  Simulation
  Operational
  Etc.

- How can requirements be elaborated to a testable level?

  e.g., Which parts to elaborate
  How to elaborate only part
  Minimal design exploration

It is clear that, for a system engineer, the nature of this type of step is essentially the same as for later design steps on the resulting requirement specification. Thus there can be no qualitative dividing line between requirements specification and design specification processes. If possible, the requirements process must be carried far enough in design to satisfy the system engineers (and the customer) that requirements are satisfactory and can be met.

The issues involved in requirement specifications are here addressed in quite general and imprecise terms just to indicate the scope of the problems. We will elaborate on many of these issues in a more formal way after we introduce the appropriate models and tools.

1.1.1.3 Design Process

The design process "starts" with some encoding of the requirements (satisfactory to the customer and systems engineer) and completes with the production of some encoding which meets design requirements and is suitable for implementation designers. Requirements may have been decomposed into relatively independent design requirements. There are six types of design steps: change of requirements, the elaboration of design decisions, the optimization of the specification, the decomposition into more manageable parts, the integration of the decomposed parts, and the interaction with other design processes. There are critical issues associated with both the initial design requirements and the resulting implementation specifications, as well as the design steps.

1.1.1.3.1 At least the following critical issues are involved in the starting and ending points of the design process:

- Is the state of design well defined?

  e.g., Is it consistent, complete, unambiguous, and testable?

- Is it suitable as input to subsequent development processes?

  e.g., Can we decode information needed for development step?

  Can the development process steps be carried out?

  Can we decide if they have been carried out?

We can not hope to give algorithms for development processes, but we can at least insist that the current state of a development process be well-defined and that we can decide when a given step has been carried out.

1.1.1.3.2 At least the following are involved in the interaction process step:

- How can decomposed systems be integrated into one system?

  e.g., Are both simply connected by interactions?

  Does one interpretively simulate the other?

  Is one translated into process of the other?

- Is the integrated system well-defined?

  e.g., Is it consistent, complete, unambiguous, and testable?

- Does integration preserve designed properties?

  e.g., If separate systems are "correct", is the integration "correct"?

  Is it at least probable that individual design decisions remain valid?

Since we will control, as part of our formal methods, the types of decompositions, we can hope to resolve these issues with algorithms for integration.

1.1.1.3.4 At least the following are involved in a decomposition step:

- Are the decomposed systems well-defined?

  e.g., Are they consistent, complete, unambiguous, and testable?

- Can they be integrated?

  e.g., Can issues above be resolved?

- Can associated non-decomposable requirements be tested?

  e.g., For consistency during decomposed design processes.

  For satisfaction after integration step.

There may exist some requirements that can not be decomposed and can only be tested at system integration time.

1.1.1.3.5 At least the following are involved in an optimization step:

- What are invariancies characterizing an equivalence class?

  e.g., What must be preserved?

- What property of members of the equivalence class is to be optimized?

  e.g., Given two equivalent members, which is preferred?

- Is it decidable if a proposed member is better than the currently designed member?

  e.g., Can we quit an optimization step because we are ahead?

We may be able to require more of some specific optimization steps. In general, we can not require less and know when the steps can be considered completed.

1.1.1.3.6 At least the following are involved in a design elaboration step:

- Is the result a well-defined system?

  e.g., Is it consistent, complete, unambiguous, and testable?

- Does it preserve validity of previous tests?

  e.g., Does validity of this system imply validity of previous system?

- What design decisions are made?

  e.g., Elaboration encodes design decision in more detailed structures.

- To what requirements are design decisions traced?

e.g., What tests must be carried out on result of
step to verify and validate decisions?

- How can the design be elaborated to a testable level?

  e.g., Which parts to elaborate
  How to elaborate only part
  Minimal implementation

Ideally, all design decisions should be testable at the completion of the design step.

1.1.1.3.7 At least the following are involved in a "change in requirements" step:

- What originated the change?

  e.g., Design decisions
  Changes in an originating requirement
  Correction of error

- What is the change?

  e.g., How defined?
  How testable?

- What is impact of change?

  e.g., Local to step
  Local to design
  Local to subsystem

- Should change be made?

  e.g., Is cure worse than disease?
  Are there alternatives?

- How can change be made?

  e.g., With minimal impact on current and completed design processes

Changes will occur. We must deal with them effectively and with minimal impact.

The design process ends when the designer has produced well-defined specifications for the systems that will collectively meet design process requirements and that can plausibly be implemented. This may require exploratory development beyond the design process prior to producing the final design. Thus we can not hope to draw neat formal lines between the various parts of the development process, but will leave decisions such as where dividing lines should be drawn for a given project to the manager who should make them.

1.1.1.4 Computer Sciences

It is clear that the computer sciences are potentially as useful to system developers as physics and mathematics are to engineers. Little of this potentiality has been realized in practice because of the complexity of current systems and because of some unique problems in scientific systems research.

The areas in computer sciences have developed only recently from application efforts in many disciplines. As a result computer sciences departments in each university have been created in differing administrative frameworks and with a wide variety of emphasis on areas of specialization. This is not surprising for a science that is so new and so potentially useful to society, as well as to the university community itself. Although there is general agreement that the area, as an academic discipline, exists quite apart from the many applications of computers to problem solving, it is only beginning to justify the use of "sciences" in its title.

There seems to be general agreement that the area of "systems" is at the heart of computer sciences. There is little agreement on what constitutes the systems area. It is clear that a substantial body of knowledge concerning the design, specification, implementation, measurement, and control of digital systems has been developed. Such knowledge is potentially useful in providing tools and application systems for all users of computers.

Because of the current lack of a consensus on the nature and standards of this new area of "systems", some special problems have arisen in exploiting research contributions in this area. Perhaps an analogy will clarify the problem. A mathematician may study artificial universes with a formal rigor that carries its own justification. A physicist may study our real universe without hope of formal rigor, and may justify his studies by the insights and control of natural phenomena. An engineer may use both mathematical and physical tools in designing applications useful to society, and may justify them by that usefulness. Computer sciences has analogs to each of the above areas. For the mathematician we have the "foundations" area, which is concerned with problems of formal systems. For the physicist we have a developing science of the design of artificial systems, which is concerned with technology--independent system universes too complex to have been formalized. For the engineer we have the implementation of systems in given technologies. We can distinguish both hardware and software engineering as subfields of systems engineering.

The justification of the foundations area is much the same as for mathematics, and the justification of the engineering areas is as usual. The system area, however, suffers from a serious problem in finding its justification. Because the system universes are artificial, we can not say (as does physics) that any insight or control into that universe is justified. Physics has a unique, self-justifying universe. The system research does not. Neither can such work be justified as mathematics, since the artificial system universes being studied are too complex (so far) to be formalized. The system researcher has a double burden. He must not only justify his solution to a problem, he must also justify the universe within which it is a solution.

Most of the contempory systems research has been carried out in the context of different, local, universes -- the locally available computer system. Each such system defines a set of constraints which creates many problems local to that system. This localization has fragmented system research into rather isolated user groups. The probems and solutions of one group are of little direct use or interest to other such groups. The informed, interested peer group to such system researchers may be very small indeed, perhaps including only local coworkers. These isolated efforts significantly contribute a more abstract (machine-independent) system universe within which a substantial community of workers may produce broadly applicable results. A universe is neither created nor justified in a day or by a few applications.

The complexity of requirements for contemporary and future system developments is so great that there is serious question of how to deal with it. Indeed, the principal result of computer sciences today is to demonstrate how impossible worst-case development processes are to specify and carry out. That is not much. We can not deal with worst-case complexity. One of the major reasons contemporary system engineers have not resolved the critical issues mentioned previously is that, in terms of arbitrary systems requirements, the critical issues have no solution. The dominant aspect is that we don't have to succeed very well to make the research worthwhile. The current costs and performance of requirements processes have enormous impact on development projects and life cycle costs. Even minor improvements that prevent errors or detect them earlier can have major implications.

The following are some very basic and obvious postulates which are often, nevertheless, ignored.

1. Don't work with either arbitrary specifications or arbitrary systems. In the worst case our problems are all intrinisically unsolvable. By imposing constraints we can restrict the domain to solvable problems.

2. Don't try to do anything that can't be done. Obviously you can't succeed that way.

3. Define the problem so that it can be solved.

4. Make it simple and testable as to whether the problem has been solved. If the problem is not simple, it won't be solved. If the possible solution is not testable, it cannot be shown whether or not the problem is solved.

We can use an evolutionary approach by embedding our new, more formal methodology in the current methodology, thus ensuring that we do not lose ground or add managerial constraints. This makes piecemeal changes possible and profitable.

We can accept, if necessary, quite severe rules that will constrain the designed system to have required properties. Not only must BMD systems work, it must also be possible to show that they work. This requirement forces us to simplify both the system design and the development process in order to make it testable. We cannot use an untestable methodology.

A practical methodology must not be computationally explosive. Most forms of analysis of complex systems are intrinsically explosive in complexity. Thus we must not use such complex procedures. We can only use relatively simple procedures. We must develop and test requirements in a way that is not theoretically complex.

We can choose the encoding of design decisions in the development process state so as to make it easy to decode them in later analysis. The designer may still use a high-level application-type language. We will work with the translator output.

By "spending" some of the potential raw computing power of distributed data processing systems, we can hope to obtain the required simplicity while still meeting performance requirements. "Shoe-horning" our design into a restricted centralized processor will only increase complexity.

We resolve these issues if we accept the above postulates, but at the risk of restricting the system domain to trivial systems. A major goal of this work is to show that we don't really lose anything essential from our system domain by accepting these postulates. Unfortunately, much work remains to be done before we reach this goal. Surprisingly, our major hope of reaching this goal rests on the complexity of the applications.

Our major task is to learn how to deal with complexity and size of systems and reduce them to manageable forms. There are two essentially different kinds of complexity we must deal with. The first is the intrinsic complexity of the attributes and relations of an object itself. The second is the extrinsic complexity of the attributes and relations of a specification of the object. Both types of complexity are formidable.

The intrinsic complexity of a BMD system is so great that we can deal with it only by accepting simplifying constraints on the design of the system itself. The questions of what constraints to accept and how to meet performance requirements are dominant issues. In addition we must be able to decide if the constraints are in fact being met by the design decisions for the BMD system. The required simplicity can make the solution of the otherwise impossible problems relatively easy.

Thus our major problems are extrinsic (that of how to define the problem structures and processes) rather than intrinsic (that of finding a solution to the defined problems). We can't possibly use most of the sophisticated analysis techniques because of combinatoric computational complexities in large complex systems. If we define our problems correctly, the solutions are not hard. We may balk a bit at some of the resulting design laws, but this will serve as a motivation for developing better design laws and better development processes.

In effect, we must approach the development process not as a mathematician (the complexity is too great for analytic solutions) but as a physical scientist interested in system invariancies. We can develop a hierarchical approach that will ensure the most basic and essential design properties and provide the foundation for future elaboration.

The formulation of such design laws provides a common basis for surveying, comparing, and evaluating current and proposed methodologies. Indeed, the most immediate impact of this study will lie in such critical evaluation and identification of potential improvements of high payoff. Ultimately, the system restrictions should make possible the creation of new development processes and tools applicable within that restricted domain. These aspects make such a research program into a winnable game. We may not solve all the problems but we can solve a useful set of them.

1.1.2 Objectives

The objectives of this proposal are to develop the following hierarchical approaches:

. to develop the "science" of design behind software engineering,

. to demonstrate the practicality of the science,

. to compare current software engineering approaches,

. to develop specifications for new design and analysis tools.

In order to reach these objectives, it is necessary to develop design schemata, required system properties, and formal models. Each of these areas will, of course, require cycles of study and elaboration into hierarchies of greater depth. No one area could be completed without commensurate studies on the others. A significant amount of work in at least the following areas will be required to meet the objectives above.

1.1.2.1 Design Schemata

The design process itself must be formalized to provide a framework within which problems can be studied and solved. The goal is to allow maximum factorization of the design process itself into independent designs, while providing a suitable level in the design schema for making all required decisions. For each design, a suitable design schema should encompass the functional, process, and implementation design problems. The formalization of such a schema requires the creation of system universe models within which decisions and laws may be defined.

1.1.2.2 Required Properties

Some properties of a particular system are valid only for that system and must be ensured by means appropriate to that system. Many required properties seriously impacting the possibility of achieving performance, integrity, evolution, and design automation can be identified and used to drive the creation of design laws and models. The hierarchical classification of these properties is essential for designers to select the level of the design laws appropriate to each design step. The certification of real-time systems alone requires a substantial number of required properties to be present. Debugging can only display errors, rather than show that there are no errors.

1.1.2.3 Formal Models

Each hierarchical design schema applied to a factored design problem will require an appropriate model and formal system for representing design laws and decisions.

It is clear that the design problems are insoluble in terms of unconstrained models. We must find the reasons for such impossible solutions and "pass" effective design laws sufficient to make them soluble. Ultimately, such laws will be incorporated in high-level design and analysis languages in order to free the designer from unnecessary details, and to ensure consistency.

The models and design laws should be sufficient to allow algorithmic analysis of design, to show presence of required properties, to allow creation of useful tools and techniques for design optimization at each level and for translation to the next level, and to reduce computational complexity of design, analysis, and transformation tools to practicality.

1.1.3 Research Plan

There are five major problems that must be addressed as a set:

- What are development process requirements in general?

. What hierarchy of system properties should be established?

. What sufficient design laws are needed to support the hierarchy?

. What development process can best incorporate the design laws?

. What encoding of the state of a development process should be used?

To limit the scope of the current work in a "top-down" fashion, we will restrict the hierarchy of system properties to be studied here to only a few, most basic ones relevant to critical issues of real-time, distributed data processing systems. We will treat the development process by first considering requirements specification and then design processes. After completion of this restricted study of the above set of problems, we will have a basis from which to address the remaining objectives. At that point we will develop a research plan to extend the relevant hierarchy of properties, to extend study to later stages of the development processes, to compare potential system engineering approaches, to develop specifications for new design and analysis tools, and to assess potential payoff for the results.

Because of the tight coupling between the potential solutions to the above problems, we cannot solve them in order. We will instead develop approximate solutions to each and iterate until all are consistent. This iteration is not completed as yet so current results must be tentative. We will do this first for the requirement specification process. Even so, we must less formally explore subsequent development process stages to assess the validity of requirements methodology. In effect we are in a development process for developing development processes.

We will develop a model for the requirements process in section 2, and a model with a detailed instance of a formal specification of the state of a development process will be presented in section 3. A brief characterization of real-time data-processing systems from the point of view of requirements specifications and relevant properties will be given in section 4. A similar treatment of distributed data processing systems is given in section 5.

With the previous results in hand, we will then turn our attention to the design process in section 6. Our current conclusions and plans for remaining work on this current contract will be discussed in section 7.

Of course, results from these later sections will have to be fed back into the requirements methodology and used to critically compare current methodologies. This has not been carried out as yet because we first needed the firmer foundation provided by this report.

1.2 <u>Specifications</u>

1.2.1 Introduction

Prior to focusing on the objectives of this work, we must introduce and structure a domain of discourse large enough to contain all systems, procedures, processes, and interfaces of interest to us. It is important that we do this without prejudicing subsequent design decisions for our development methodology. Such an extrinsic structure is clearly not unique, and is justified only as a sufficient framework for developing and discussing this work. Modifications, elaborations, and evolution of this descriptive framework are to be expected in light of subsequent work.

1.2.2 Specification Levels

1.2.2.1 Introduction

We will first identify and define several kinds of "things" we will need to discuss. We will order them (as levels) by their degree of abstraction. One <u>level of abstraction</u> is lower than another if it contains all of the information (and more) of the other. The highest level of abstraction thus encodes the least number of design decisions. We are not here critically evaluating these levels for our purpose. That will come later. We are only defining and ordering them here. We will first draw a distinction between a specification and a description.

A <u>specification</u> of a "thing" defines all of its attributes and their values that are of interest. Any "thing" having those attributes and values is considered to be one of the specified "things".

A <u>description</u> of a "thing" defines some of its attributes and their values. Any "thing" having those attributes and values may (but need not) be considered one of the specified "things". A description may be missing the definition of some attributes and/or values which are important and may contain some that are not important.

1.2.2.2 Interfaces

A collection of interacting systems may be characterized, with respect to their interactions, by specifying their interface. An <u>interface</u> can be defined by a relation between all system outputs to all system inputs for the entire collection of interacting systems. Both the domain and the range of an interface relation are potentially infinite sets of potentially infinite time sequences of messages.

The most abstract level of specification of anything is an interface, since no internal structure of the interacting entities is defined by the interface specification. A formal treatment of interface relations and some results on various forms of relaxed equivalence have been made by P.Z. Smith [Smi76].

[Smi76] Smith, P. Z., <u>Functional Equivalence of Parallel Processes</u>, PhD Thesis, University of Wisconsin, 1976.

Even in theory, interface specifications are difficult if not impossible to encode in a finite and usable manner. In practice, they are so impractical as to be irrelevant in all but very special circumstances. However interface descriptions of many kinds have been used in development processes and have played an important role.

1.2.2.3 Processes

The concept of process is central to most of this work. Many kinds of processes have been defined including continuous, stochastic, and discrete processes. We will include continuous processes only via the specifications of their simulations as stochastic or discrete processes. We will use stochastic processes to model dynamic properties of our discrete specifications and will define them when appropriate for our methodology.

A <u>discrete process</u> is a possibly infinite set of possibly infinite sequences of states. Each sequence of states is a <u>computation</u> of the process. A <u>process step</u> is the transition from a state in a computation to the successor state in that computation. A <u>state space</u> defines the set of all possible states of a process.

A discrete process is <u>deterministic</u> if and only if the immediate successor state of each state is unique. If any one of a set of states may be the immediate successor state of some state, then the discrete process is <u>nondeterministic</u>.

Any discrete process of interest to us may be specified by a state space, a set of initial states, and a state successor relation. A state successor relation can generate a computation of the process by its application first to an initial state and then successively to its previous results. A computation of a discrete process can thus be interpreted as a time sequence of states. Each state occurs at some time prior to a successor state. The most recently produced state is the current state of the computation (or process).

One source of non-determinism arises from potential interactions between discrete processes whose computations occur asynchronously in time. A cluster of discrete processes is a set of potentially interacting discrete processes. The nature of the interactions of the contained discrete processes can be defined by asynchronously applied relations on the whole cluster. We will develop a formal treatment of clusters in a later section and discuss a number of relevant results.

A discrete process is a lower level abstraction than is an interface, since some additional constraints on the computations have been specified beyond those implicit in an interface specification. An interface specification does not, for example, constrain the state space or the sequence of steps of the process. However, the concepts of "variable" and of "assignment of value" to variables are not specified at the process level. States literally represent the arguments of the state successor function; the only replacement specified is the substitution of the entire successor state for the current state of the process.

1.2.2.4 Procedures and Interpreters

We can go further in decreasing abstraction by introducing the concepts of variable and assignment, encoding them as procedures and address spaces over which names of variables can be decoded. A procedure may be used to generate a computation by using it to define an initial state of an interpreting process. A conventional programmable digital system can be initialized to contain a procedure and carry out the computations of the interpreting process. The state successor function of the interpreting process thus defines the semantics of the procedure while the form of the encoding in the interpreting process state defines the syntax of the procedure. The set of variable values defined in the current state of the interpreting process defines the current <u>procedure</u> <u>state</u>.

1.2.3. Encodings

1.2.3.1 Introduction

Another dimension to explore in specifications is that of the encoding of the specifications. This is also a critical area since ultimately we must be able to analyse our developed specifications. Because we cannot decode arbitrary encodings, subsequent analysis may become impossible unless we carefully encode so as to allow decoding.

1.2.3.2 Encoding Complexity

The two basic ways to encode information are in the structure of an object, and in the values of attributes of the object. The <u>object structure</u> is an explicit part of the object that is invariant to a set of transformations of the object. Encoding in structure thus makes decoding explicit and relatively easy. Encoding in values is implicit since decoding requires a study of the interpreting process. Because this value decoding is so intrinsically complex, in general we must avoid value encoding of any information we must subsequently recover from that encoding. For example, optimization should always be defined as a structural transformation rather than as value transformations. A corollary is that processes rather than procedures should be optimized since the process structure of the former has been translated into values in the latter. Our encoding must thus provide a very rich set of structures.

1.2.3.3 Formality

An informal specification is a contradiction in terms, (really a description at best), and may neither precisely specify attribute values nor even specify which attributes are of interest. Formal questions of completeness and consistency of both the informal specifications and the things being specified have no formal or precise answer.

Informal specifications are like value encoding in that only an interpreting process can decode them. Because specifications in English require an interpreter of English (a human being) to decode them, we can not automate such interpretations. In addition, it is very difficult to specify precisely in English, where ambiguity and context sensitivity are the rule rather than the exception. Another informal specification is the thing being specified. The thing literally but informally specifies itself since it does not specify which attributes and values are of interest. In the limit, all things are uniquely themselves and do not specify any other thing. In a degenerate sense we can treat the literal thing as our last (and informal) specification of the state of our development process.

A _formal specification_ will precisely specify both attributes and values of the thing being specified. Whether a thing is a formal specification or not is decideable. An _effective specification_ is a formal specification that can be interpreted to display the attributes and values of the thing being specified. An effective specification may thus be tested for completeness and consistency of the thing being specified.

1.2.3.4 Functional Specifications

1.2.3.4.1 Introduction

We will use the word "function" to denote a mathematical function in this work. While a <u>nondeterministic</u> <u>function</u> is more precisely a relation, it will be convenient to use the unqualified "function" as being either deterministic or non-deterministic depending on whether its range elements are values or sets of values.

The concept of function, defined and extensively studied by mathematicians, is a standard form for encoding formal information. Functions may be defined implicitly, explicitly, or as primitives.

1.2.3.4.2 Implicit

Implicitly (axiomatically or by inference) defined functions may be studied by either an interpretation of the definition or as a solution to the defining axioms or inference. Such functions thus provide little or no explicit structure for encoding. It may not be possible to even decide if there is such a function from implicit definition.

1.2.3.4.3 Explicit

Explicitly defined functions may be studied for their structure as well as their value. The structure may be encoded as various forms of synthesis of more primitive functions. We can bring most of mathematics to bear on the specification problems by using this encoding.

1.2.3.4.4 Primitive Functions

Primitive functions may be undefined, described, or defined implicitly. The primitive may be undefined either because the specification is incomplete or because any function will formally meet the specification. A description of a primitive function may be interpreted as constraining the choice of functions to be used in defining the primitive. Thus ultimately, explicitly defined functions rest on implicitly defined, informally described, or undefined primitives.

1.2.4 Applications

1.2.4.1 Introduction

We have been careful to distinguish between a specification and the thing being specified. There is another important distinction to be made -- that between the thing being specified and what it is interpreted (e.g., by the specifier) as representing. While our formal specification methodology is independent of such interpretations, our development process is not. Three important interpretations of a specification are as a problem, as a solution, and as an implementation.

1.2.4.2 Problem Specification

Development processes frequently are carried out with an informal problem description that prevents any precise determination of whether a solution to the problem has been found. Although our methodology must support such development processes, it can not exploit such informal knowledge of the problem in optimizing and testing design decisions. The development of the problem specification will include much of both the system design and requirements engineering effort, and thus must be addressed by our methodology. A control system problem specification must define the required behavior of the things being controlled.

Each level of formal specifications, as discussed previously, may be used to define the problem. A level so abstract that no one can discover if a solution exists or what the solution might be, may be unsuitable for starting a particular development process. The problem specification will more likely be started at a high level and developed to a low level of abstraction as more detail of the problem is encoded. A brief description of a problem at each of the levels is given below.

An interface-level problem specification might define a threat (a set of attacking missiles) and a defense (a set of interceptor missiles) as a time series of interactions between them. The problem would be to find and implement such observation and control systems as to cause the interface specifications to be met. The often-used scenarios, if presented as formal specifications, would be an example of such an interface-level problem specification.

A process-level problem specification might define the required movements of threat and defense missiles as interacting discrete processes as in a "real world" model. The elaboration of their state successor functions and their interactions defines the requirements of the problem. It is at this level that many of the basic system design trade-offs, in terms of problem parameters, are made or delimited. The process specifications should be effective so that the consequences of such trade-offs on problem behavior can be explored. Differential and algebraic equations can be used in an effective functional specification of the required "real world" processes.

A procedure-and-interpreter-level problem specification might define a "real world" simulator (interpreter), and define each threat and defense missile as a procedure that carries out the required behavior. Such procedures do not necessarily solve the problem since they can be written with perfect knowledge and control of events in the real world, and thus may not model an implementable reality.

A problem specification may be developed through several levels of abstraction. Requirements are now expressed in terms of the formal specifications. As their relationships and design trade-offs are then implicit in the specifications, they can be addressed by our methodolgy.

1.2.4.3 Solution Specifications

The results of a given phase constitute a "problem" specification for the next phase of the development process. The solution specification thus becomes the implementers' problem. Indeed, for simple problems implementers may accept the problem specification and create a solution without any formally recognizable solution development phase. Given the problem, some might successfully just start writing the programs that solve it. However, because of the complexity of both potential design trade-offs and the solution processes, discovering the suitable processes based on valid real-world assumptions (e.g., imperfect knowledge of the threat) may require a significant phase itself. A solution phase must develop a sufficient level of detail in the required processes as to both satisfy the problem specifications and be feasible to implement. Solution specifications could be at any level of abstraction, but will normally result in a process- or procedure-level specification because it must show that it does solve the problem. Such a demonstration at the interface level would be very difficult. The suitability of a solution for implementation is also easier to assess as we go to lower levels of abstraction. In practice, the end of the solution phase will almost certainly be some form of process specification.

1.2.4.4 Implementation Specification

An implementation consists of procedures and their associated interpreters that together generate the computations of the solution processes. Their design may involve both software and hardware design decisions. The implementation phase accepts the solution process specifications and produces the implementation specifications. Clearly there may be subsequent phases of construction, operation, etc. However, we will not be much concerned with those phases here.

An implementation specification could be at any level of abstraction, but will normally be at the procedure/-interpreter-level. An implementation specification must be testable for both correctly supporting the solution processes and meeting performance criteria. It is only at the procedure/interpreter level that metrics for many (interpreter) resources can be established. Many of the required attribute values are not defined until that level, and thus performance can only be forecast at higher levels.

1.3 Development Processes

1.3.1 Introduction

There are a number of distinctly different processes involved in a BMD system development. The three major processes of interest are the specification, evolution, and management processes, in that order. The development of methodologies to support such processes must also occur in that order. We can not provide an automatable methodology without a precise model for specifications. We can not develop specification processes without a methodology to carry out process steps. We can not define evolution processes except in the context of specification processes. Management processes should be developed in the context of both specification and evolution processes so as to make the management feasible and well-defined. All such processes, however, have some common aspects that we will discuss in this section. These include the concepts of discrete processes, methodologies, and phases.

1.3.2 Discrete Processes

A discrete process is a sequence of well-defined states produced by the application of some process step procedure. The procedure need not in general be automated, algorithmic, effective, or well-defined. The minimal constraints on a discrete process are that the states be formally well-defined, and that there be a formal test for the completion of each potential step of the process. The primitive components of a well-defined state may be as high-level and abstract as desired.

The specification of the state of a process may take any of the forms discussed in the previous section. These forms are by no means equivalent for our purposes. For example, suppose we wished to validate a state specification. This might be done by computation with procedural specifications, by stochastic simulation with process specifications, or by analysis with axiomatic specifications. For a given process, axiomatic and interface validation may be impossible, functional validation may be insufficient, and procedural computations may be impractical. Clearly, we must be prepared to accept severe constraints on the specification methodology.

We must define (in order to deal with complexity) at least the following types of process steps:

.<u>Initialization step</u>

The initial state of a process is created by the initialization procedure without a process state input.

.<u>Decomposition step</u>

When the attributes of a process state are loosely coupled with respect to process design decisions, a state may be factored into initial states for independent design processes.

.<u>Integration step</u>

The states resulting from the decomposed processes are integrated into a single initial state for the subsequent steps.

.<u>Interaction step</u>

When the process state attributes are partially coupled with respect to process design decisions, then a decomposition step may produce design processes that must interact during some of their steps.

.<u>Partitioning step</u>

The process state may be factored into interacting components whose individual operations are but loosely coupled via the component interactions.

.<u>Transformation step</u>

The primitive structures and values of the state are transformed.

The most essential property of these processes is that they can model the computations of a system, the evolution of a system, the development of a system, and the specification of a system. By developing our methodology in this context we may hope to apply elements of it to many different processes.

1.3.3 Phases

A <u>phase</u> of a process is characterized by the following properties of the terminal state of that phase:

.The required tests can be carried out by the phase "contractor" and validated independently. The work is not done until the tests above are passed. Thus the phase must produce input suitable to the testing methodology.

.The specified system satisfies the "customer" of the phase. This implies that the specification is testable for <u>specified</u> axiomatic invariancies, behaviors, functional properties, procedures, and implementations as required to satisfy the customer that the phase has been completed, and that a system meeting the specification can be built.

.The specified system satisfies the "contractor" of the next phase. This implies that the specification is testable for its suitability as input to the following phase (i.e., the contractor can plausible carry out the next phase).

Any section of the development process for which a suitable testing methodology can be developed is a potential phase. Not all potential phases are necessarily cost-effective because of the expense of testing. The management of the development process may still require the potential phase testing in order to measure progress. Ideally, our development process will be testable at each step, so that any desired section may be made a phase by management decree. We would thus free management to manage without unnecessary constraints from the methodology. In particular, Military Standard documents could be produced at any desired point, and phase definitions can be adjusted as required by management.

We will postulate the sectioning of the development process into requirements, design, implementation, and operational phases. This implies that we must also develop an appropriate methodology for testing each phase, in addition to the methodology for developing the steps of that phase. This postulate is not trivial and makes the requirements process into a phase of the development cycle. Some system developers will claim that the problem is inherently so complex that the required phase testing cannot be done. Indeed, some system requirements may not be testable short of operational use, thus requiring some judgement risk at intermediate phases. We will accept this postulate as a basis for research because of the importance of the requirements phase.

The nature of the tests for each phase requires procedures which forecast the results of subsequent phases. These forecasting procedures may be carried out as specialized forms of the subsequent phases that exploit the methodology of those phases. Thus the methodology of each phase may be partially included in that of the previous phase. The allocation of personnel trained in the methodology of a given phase to previous phases is simply a management decision. Management could train personnel more broadly if desired.

1.3.4 Methodologies

A <u>methodology</u> can be defined as a set of procedures, rules, and tools that support a family of processes. The procedures are used as steps of the process. The tools are used to carry out the various procedures. The rules are used to decide which steps should be carried out.

The rules will encode experiences as to good step sequences and constraints required to ensure certain properties of the process results. In the limit, these rules can be axioms, validated by theory and practice, which are sufficient to ensure some required properties.

The procedures encode possibly non-discrete or non-observable operations which will lead to the subsequent state. In the limit these procedures may incorporate a set of decisions sufficient to eliminate need for further rules.

The tools encode effective (i.e., automatable) operations that are part of a methodology procedure.

For some processes, a theory can be developed that allows a sufficient set of axioms to be exploited in a tool for automatically carrying out the process. We cannot hope to do this for all processes, but experience has shown that even partial theories can have a powerful impact on methodologies and are well worth developing and validating.

The design of the methodology of a phase is subject to constraints from the subsequent phases. These constraints arise primarily because subsequent tests may not otherwise be possible. System complexities are normally so great that arbitrary (worst case) systems are not testable for required behavior. We must accept constraints on the domain of system solutions as well as constraints on the way they are specified as the price of being able to decide that the requirements have been met.

The acceptance of such constraints (however severe) is implied by the acceptance of the postulated phases of the previous section. Given a set of sufficient constraints, we can certainly look for ways to relax them in order to extend the methodology. If the resulting methodology is only partially useful, it may still be the best we can do currently.

If our developed methodology is to be used, it must:

1. not constrain use of any other procedures that developers may choose when our methodology is inadequate,
2. not unnecessarily constrain evolution of the methodology, and
3. include tools to measure the impact of the methodology on the development process.

Because the complexity of our problem is so impressive, we will be able to find "solutions" only if we obey the above constraints. Our methodology will not at first (and probably not even at last) address all required system properties, and so we must not complicate the problems of addressing the other issues. If we can't help on a given issue, at least we must not interfere. If there is only a "point solution" to the methodology, we will probably never discover it. We <u>can</u> find a methodology that works with respect to some properties. Then we can evolve improved and extended methodologies by experience, experiments, and theory.

With sufficient experience we may evolve theories that will allow us to eliminate parts of the methodology and automate more of the process.

2. REQUIREMENTS SPECIFICATIONS

2.Ø Background

The BMDATC has long recognized the crucial role of systems developmnt methodology in meeting the requirements of BMD applications. A substantial program to develop a current state-of-the-art methodology has been carried out to produce a baseline methodology. However, the lack of a sufficient science of design in general and of requirements on the specification of system requirements in particular has impaired attempts to evaluate the results of such programs in order to guide future methodology developments and applications.

The enormous expense and the slow development time of BMD applications make it impractical to track changes in threats, missions, and technology at the operational level. We would obtain enormous cost benefits and significantly shorten deployment time, when such deployment is decided, if such changes could be analyzed at the requirements level. However, while requirement specifications consisting of large informally specified documents may be testable for some properties, they do not provide a sufficient basis on which to judge whether the required systems can be designed, will behave properly, will be feasible to build, and will fulfill the mission.

Data processing systems requirements have frequently been specified by default, i.e., they should control so as to fulfill the mission. How can we use such default requirements (or what alternatives can we propose) to develop formal requirements which are suitable to data processing system design? Further, how can we use the potential of distributed data processing throughput to get the simplicity required to enable us to test each phase of development? These and other questions require a much more fundamental and scientific study of the critical problems involved in extending the current methodology in requirements specifications.

We need a model for requirements development processes in order to identify the needs and mutual constraints on the related methodologies. The rudiments of such a model are developed in the following subsections.

2.1 Introduction

The development of requirements is usually an intrinsically complex problem that may never be solved even in principle. We can not present nor hope to develop the requirements development process. Moreover, since there may be no complete solution, we can not even develop a uniquely best methodology for supporting the development of requirements.

2.1.1 Purpose

We can develop a reseach plan for the extension and improvement of current methodologies, and characterize the natures of the proposed changes and their potential impact on the problems of BMD methodology. We will first discuss some methodology concepts. The current BMD methodology requirements have been summarized by Davis and Vick [DaV77], a portion of which is quoted here.

> "The results from previously mentioned BMD developments and other large scale weapons systems experience resulted in the identification in the early 1970's of the following set of characteristics which a software development approach for BMD must have. ... The required characteristics include:
>
> .Data Processing Description Capability. The system must allow for inclusion of data processing limitations early in the development. This must include the means for assessment of data processing induced system limitations (e.g., processing delays and inaccuracies) as well as the ability to provide accurate estimation of the data processing hardware requirements, and support tradeoffs between alternative approaches.
>
> .Requirements Orientation. Requirements approaches must be developed which insure means for stating the required processing without inclusion of unwarranted design detail; insure unambiguous communication of intent; provide a means to validate requirements; insure their feasibility; and be responsive to the invariable change.
>
> .Design. The software design process must provide a means for earlier error detection, rapid modification, and designed-in reliability. The approach must insure the production of a highly reliable modular product which will minimize the life cycle costs.
>
> .Automation. The system must possess as much automation as possible in every phase of software development. The aids should be such that they provide maximum utilization of the thoroughness of the computer to eliminate many sources of human error.
>
> .Management. The system must consist of well defined phases containing intermediate milestones which provide for measurements and evaluation of progress. Techniques must be devised which allow

[DaV77] Davis, C. G., and Vick, C.R., "The Software Development System," IEEE Transactions on Software Engineering, January 1977.

a priori costing and scheduling based upon a defined, structured approach to development.

.Testing. The system must provide means for the allocation of performance to the data processing subsystem, the refinement of that allocation and improved means for the testing, verification and validation of that performance as an integral part of the development cycle.

.Structured Decomposition and Development. There must be allocation and improved means for the testing, verification and a technology which forces the problem to be stated and structured at a high level, analyzed at that level and then allows the developer to proceed with the addition of detail in an orderly, defined and measurable fashion. This must proceed from early system definition through code delivery in a traceable and flexible manner. This technology must assure maximum designed-in reliability in the development cycle."

These requirements, however valid, give us no immediate handle on how to develop a methodology to meet them.

2.1.2 Research Plan

We will now describe a statement of work for a research program to address the requirements methodology issues. This current work only partially addresses the required tasks and should be considered as an exploratory effort.

2.1.2.1 Objectives

Our major goal is to create a methodology that will support the tracking of threat, mission, and technology changes at the requirements level while still providing confidence in the behavior, feasibility, and effectiveness of the required system.

It is expected that this program will involve a five year effort. The high risk involved must be balanced by potentially high payoffs and early conceptual development and validation.

The general form of a requirements methodology research process can be described by the following steps:

- Identify critical BMD development problems
- Identify potential requirements methodology improvements
- Assess impact of improvements on critical problems
- Select high payoff improvements
- Develop research plans (requirements)
- Develop acceptance test plans (for research results)
- Carry out planned research and tests
- Evaluate actual impact
- Extend "production" methodology.

The short-term program will be a one-year effort that should develop the concepts, theories, and principles required for development process methodology extensions with a potentially high impact on critical BMD problems.

A set of critical BMD problems that can plausibly be addressed via methodology extensions will be defined precisely. Potential methodology extensions will be identified and assessed for impact on the critical problems. We are more interested in solutions to problems (even if high risk programs are required) than in mere amelioration of some problem symptoms. The research required to define problems, methodology extensions, and sufficient conditions under which the defined methodology extensions will solve the defined critical problems will be carried far enough to produce plausible development and acceptance plans for the selected methodology extensions.

Sufficient exploratory work on the methodology extensions will be carried out to support the plausibility of the resulting methodology proposals.

It should be recognized that a significant part of the methodology extension problem is extrinsic and lies in the structuring of both problem and solution definitions so that useful sufficient conditions can be found. The first year's work should be the key to the entire program and should result in a plausible assessment of the benefits of the subsequent program.

2.1.2.2 Requirements

The following tasks should produce the following documentation:

- Critical BMD problem definitions
- Methodology extension definitions
- Problem impact and payoff assessment
- Design principles and sufficient conditions such that each extension can be developed and will have the desired payoff
- Development plan for each extension
- Acceptance test plan for each extension

The methodology extensions proposed and evaluated should be relevant to the axiomatic and requirements phases previously described.

2.1.2.2.1 Task 1: Identify Critical BMD Development Problems

Describe a sufficient model for the BMD development process within which the critical BMD problems can be specified formally and precisely. This task will require development of both system and specification formalisms suitable for the remaining tasks. Sufficient conditions will be developed such that it is possible to decide if a proposed solution solves the problem. These conditions will be used as laws for subsequent tasks.

The BMD problem areas studied will extend over the entire development process and will include (but not be limited to) the following aspects:

- Design trade-offs
- Specifications
- Evolution of specifications
- Measurements and analysis of specified systems
- Forecasting performance
- Testing performance and readiness

2.1.2.2.2 Task 2: Identify Potential Methodology Extensions

The current state of the art should be used as a baseline for improvements. This task should find the limits on our current knowledge and assess their impact on development methodology. Research required to relay these limits should be carried out and evaluated in terms of the potential new methodology extensions based on the new knowledge.

2.1.2.2.3 Task 3: Assess Impact of Extensions on Critical Problems

This analysis should, for each proposed extension, identify the BMD problems addressed, show the relevance of the extension to solving the problem, and assess the potential payoff. The high payoff extensions should be pursued in the remaining tasks, even if high risk research programs will be required. The requirements in the methodology extension in order to provide the high payoff should be identified.

2.1.2.2.4 Task 4: Develop Research Plans

Research plans to develop the potential methodology extensions will be developed and elaborated to meet the short term objectives in Section 2.1.2.1. These plans must contain a specification of the requirements to be met by the proposed effort, and a plausible approach to meeting them, as well as estimates of the resources required.

2.1.2.2.5 Task 5: Develop Acceptance Test Plans

The plans of Section 2.1.2.2.4 should be testable, and tested, for consistency with the design laws and requirements developed in the previous tasks. An acceptance test plan to decide its success must be developed.

2.1.2.3 Industrial Base

The appropriate bidders for such basic research will presumably come from universities or other non-profit research organizations. The intent is to award three contracts to maximize the scope of the concepts to be considered, and to facilitate the collaboration of the contractors. Since many potential research contractors may be unfamiliar with current BMD system engineering methodology, the government should provide the services of such a systems engineer at workshops and technical direction meetings.

2.1.3 Requirements Processes

We can now specialize our discrete processes and methodologies for requirements processes. This specialization will give us a more structured process for specifically studying BMD problems. Unfortunately, we will encounter many requirements issues that must be resolved prior to BMD specialization. This is because in many important aspects, the BMD requirements process raises the most severe cases of problems that appear in a less severe (but still unresolved) form in many other systems developments.

The states in the requirements process will be interpreted as requirements specifications.

2.1.3.1 Minimal Requirements Processes

The actual step sequences selected by requirements designers are sensitive to the methodology, the organizations, the contracts, the system being developed, the technology, and almost everything else, including personal idiosyncracies of various managers. Experience with a methodology on a class of problems may later lead to acceptance of rules constraining such sensitivities, but we must first address the methodology problem.

The requirements process can not be considered in isolation from the remainder of the development process, since constraints arise from all phases of development. Not all problems can be solved in the requirements phase, but many can only become solvable in later phases if the requirement specifications are suitably constrained.

2.1.3.2 Concept Formulation

The initialization step of the requirements phase is potentially the most important and complex step of the entire development process. It is also the most complex step for a methodology and hard to automate. Even people have difficulty in carrying out this step.

2.1.3.3 Requirements Testing

The purpose of a requirements specification is to encode requirement design decisions in a testable way. Thus the primary source of specification design constraints lies in the tests required. Given an arbitrary test, we can not even in principle create specifications that can be shown to pass the test. Thus we must choose our specification representation so as to maximize the accessibility of encoded information and allow the broadest possible domain for testing. The requirements designers must either adapt their requirements process to the tests in our methodology, invent new ones, or fake it and have faith, hope, and charity. We can divide such tests into tests of the specification itself and tests of the system being specified. This corresponds to the usual concepts of syntactic and semantic testing.

2.1.3.3.1 Specification Tests (Syntactic)

Syntactic tests are made on the form of a specification and are not sensitive to the meaning of the primitive concepts encoded in that specification. In order to be testable the specifications must have at least the following properties.

- Formal

  The tests must be well-defined and the results decidable in order to automate test and procedures.
- Complete

  They do specify a requirements process state.
- Unambiguous

  The question "what is specified?" is decidable down to the level of the primitive concepts.
- Consistent

  Given an ideal technology, there could exist a system satisfying the specifications.
- Effective

  There are procedures for the tests that can be carried out.
- Practical

  The tests can be made with practical amounts of test resources.

This list is not complete and the division of tests into specification and system tests is not precise. Many tests may fall in both areas.

The requirements methodology will incorporate a potentially large set of rules which, if followed, will ensure that the specified system will have certain desired or required properties. One of the more effective ways to exploit and test these rules is to encode the resulting design decisions in the form (rather than the content) of the specification, so that specification (syntactic) testing will suffice. There exists an enormous body of information and set of automated tools for such testing. Our specification representation must facilitate such encoding.

2.1.3.3.2 System Tests (Semantic)

Semantic tests of specifications involve the meaning of the primitive concepts of the specification. Thus some interpretation of the specification is required to carry out the test. This interpretation must also be formal, complete, unambiguous, consistent, effective, and practical if such tests are to be part of our methodology.

For the development process, it is sufficient and convenient to define the states procedurally in the form of a language-parsing procedure and a grammar. It is this "state language" whose syntax and semantics are being tested by our methodology. We are not here much concerned with the syntax of such a language (in the current state of the art there are few constraints on parsing), but we are concerned with the interpretation of its semantics.

The state language semantics must be interpretable as specifying a set of interacting digital systems. For our purposes only the functional specification is suitable. Axiomatic specifications will not be effectively interpretable. Interface specifications cannot be practically interpretable. Procedural or implementational specifications are too elaborate and detailed and are only obtained at the end of the developmental process. Functional specifications, however, have many very useful properties including mathematical analysis and multi-level abstractions. Indeed, such functional specifications can be used throughout the specification development process.

We will interpret such functional semantics as specifying a system by defining the computational processes of that system. This is sufficient for testing of specifications since the interpretation is effective. We have not foreclosed our methodology from specifying non-data-processing systems since the semantically specified system may _be_ a digital model of the required system. We have implied that we can (and must) functionally specify asynchronously interacting parallel processes. At least one way has been described [Fit76].

The functional specification of the system must be effective in the sense that, given the specification, we may interpret it to provide any observations of the system process to the level of the primitive functions of the specification. This interpretation process provides the basis for semantic testing of the specification.

However great the complexity of a function definition, mathematics has already defined its interpretation as a mapping of values. Primitive functions simply have

[Fit76] Fitzwater, D. R., "The Formal Design and Analysis of Distributed Data Processing Systems," University of Wisconsin Computer Science Department, Report CSTR 279, October 1976.

non-functional specifications of which value in the domain goes to which value in the range. Thus, we cannot functionally test primitive functions except for questions of domain and range.

We can use a variety of interpretations of a primitive that will simulate the behavior of that primitive to the desired detail: these interpretations include procedural approximations, symbolic execution, operations on sets of values, and stochastic mappings. Indeed, a functional specification is as intrinsically analysable as any specification could be.

The observed behavior or analysis results can be compared with the requirements of the axiomatic model. With a detailed axiomatic model, substantial automation of the semantic tests may be carried out.

2.1.3.3.3 Forecasting

In addition to testing the behavior of the specified system, we must be able to forecast the feasibility of the requirements. This implies that both the customer and the next phase contractor must be satisfied that the requirements (including performance) can be met during subsequent development phases. Of course, no forecasting methodology is going to be totally accurate, so iteration between phases (however unfortunate and costly) cannot be absolutely prevented. However, our methodology may reduce the cost of such iteration and may be usable by the current-phase contractors. The formalization of the specifications plays an important role in tracing and changing requirements as discussed in the next section.

Forecasting procedures for data processing systems are not well developed. The most powerful current technique is hard-earned experience in similar systems. Indeed, if one can manage an evolutionary approach (a sequence of similar but evolving developments), such previous experience may be the best possible forecasting method.

By using a relatively homogeneous methodology throughout the development process, we can allow the requirements process to explore critical issues down through design and implementation phases as needed to satisfy the requirements test phase. Full scale or complete developments may not be needed.

2.1.3.3.4 Validation and Verification

The validation and verification methodology now simply contains the testing tools required for the development process. There may be real reasons (management decision) to have independent V & V contractors, and they may either carry out the testing process of each development phase together or do it redundantly and independently for confidence. These are all management decisions with little impact on the testing methodology. The primary impact is an enhanced need to use the formalized specification as a communication medium so that such management decisions are feasible, and so the required behavior is defined.

2.1.3.4 Changes

Changes in requirements during the development process are a serious problem that our methodology must address. Changes may arise during all development phases due to changes in mission or technology, correction of errors, discovery of unpredicted consequences of previous decisions, or the use of an evolutionary methodology.

The management of change is an important aspect of the development process, and can become quite complex while juggling schedules, resources, system requirements, etc. This problem is formidable when the system being changed is not precisely specified. Our methodology may be able to minimize the impact of changes arising from all but mission and technology changes by making them a well-defined part of the development process. Since we can determine precisely what is changed and what is affected by that change, our entire methodology may assist in testing and implementing the change. With substantial automation of development procedures and analysis of specifications made possible by our formalization of specifications, fewer errors and more prediction may occur and make earlier changes feasible. The time required to assess change impact and to carry it out may be drastically shortened.

If we accept methodology constraints required to support evolutionary processes for each development phase (including the operational one), we can develop a methodology for change that may make evolution the normal development process. The practicality of such a methodology is strongly dependent on the formalization of sufficient attributes of the functional specification. We need to find something like the Taylor series expansion about a point to simplify the step of evolving a new system from an old. Much research needs to be done in this area. Simple "modularity" is not enough. We must control both the structural and the dynamic aspects of changes in our evolutionary processes. Evolution at the requirements level is a very attractive concept for BMD systems, but will require a substantial and trustworthy methodology built on possibly severe system constraints to achieve the required simplicity.

2.1.3.5 Language

The current state of the art in language design is quite advanced and is not, itself, either a serious problem or a research topic for requirements methodology. The problem is semantic (what to say) rather than syntactic (how it is expressed) since a translator can provide the user with his desired forms of expression while producing from them the appropriate results. Consequently, we can develop our methodology without consideration of language constraints. We can then develop the appropriate languages for defining states and procedures while incorporating the current rules without unduly stressing language development methodology. Languages and translators thus become development projects in the context of a given methodology. In the absence of a methodology, language and translator development becomes non-critical and premature. Requirements methodology must drive language design rather than the other way around.

2.2 Initialization

The initial system concepts will inevitably be informally (possibly inconsistently and certainly incompletely) specified. Informal procedures may be used to test and elaborate such specifications. The only assistance our methodology could provide would lie in computations, *ad hoc* simulations, management of a data base, document generation, and change control (in short, all of the services that are currently supplied in one form or another). Clearly, current methodologies can be exploited more than they have been, and can always be improved with experience. These things can and are being done in developmental programs and in the context of particular applications.

Our methodology can provide further assistance only at the cost of precisely specifying the initial requirements state so that our methodology is then applicable. The precisely specified state might of course be only an approximation to the desired state. At least, the methodology for the subsequent requirements process will support the exploration of the implications of concept decisions. We would thus expect substantial iteration of at least the earlier steps of the requirements process.

So far we have not specialized our approach to distinguish between system and subsystem level requirements. If the initial application of our methodology is at the subsystem level and it has not been previously applied at the system level, then the above initialization procedures are still relevant. If the system-level requirements were developed with our methodology, then by definition the suitable subsystem requirements initial state will be part of the system-level requirements specification.

2.2.1 Problem Phase

The leap from the informal problem specifications to the formal requirements initial state is an approximation. We can postulate a problem development phase to partially bridge this large gap. The terminal state of the problem phase becomes a more precise and formalized specification of the original informal starting specifications.

First of all the input to the requirements process -- that vague, informal idea of "desired behavior" -- must be formalized to the level where it can be tested. This can involve having an axiomatic specification, i.e., a model of the real world in which axioms describe the behavior of objects in the real world. These axioms can be mathematical equations based on "perfect knowledge" of all the relevant variables in the system. For example, the new position and velocity of an interceptor may be dependent on its previous position and velocity, and on the position and velocity of the threat missile it is seeking, combined with invariencies such as the laws of physics, the physical characteristics of the missile, etc. The axioms must be elaborated to the level of detail where they can be tested to determine if they produce the desired behavior. If they do not, either the idea of the desired behavior must be modified, or the axioms must be modified so that they approximate more closely the desired behavior.

It is up to the user to determine how much or how little he wishes to put into this axiomatic problem specification. If the entire "desired behavior" can be modelled by the axioms, then the testing of the effective model of the BMD system can be highly automated. On the other hand, if the user wishes to leave the "desired behavior" as a totally informal notion, then there can be less help in automatic testing, checking for consistency, etc.

First the inputs and outputs of the axiomatic-world process must be defined. The input will be the user's ideas of the desired behavior of the objects in the real world -- vague, informal, incomplete, possibly inconsistent or infeasible; for conciseness, this notion will be called a "cloud". The output of this process will be a set of axioms which describe the desired behavior of objects, and which are precise, formal, consistent, and as complete as the user wants to make them.

There are three procedures:

GEN-AX-SET: CLOUD --> AXIOM-SET

This procedure generates a testable axiom set which is consistent with the cloud and the real-world laws.

ELAB-AX-SET: CLOUD x AXIOM-SET --> AXIOM-SET

This procedure generates a new axiom set which is a testable elaboration of the input axiom set, consistent with the input axiom set, the cloud, and the real-world laws.

IMPROVE-CLOUD: CLOUD --> CLOUD

This procedure elaborates or changes the input cloud to create a new cloud.

Figure 2 is an example of a process built from these three procedures to generate an axiom set from a cloud.

The extent to which "perfect knowledge" can simplify the axiomatic specification for BMD systems is speculative. If this could be done in sufficient detail, the addition of a particular threat might allow systematic validation of the ideal system specifications and still earlier feedback in the concept formulation phase. Such a specification could

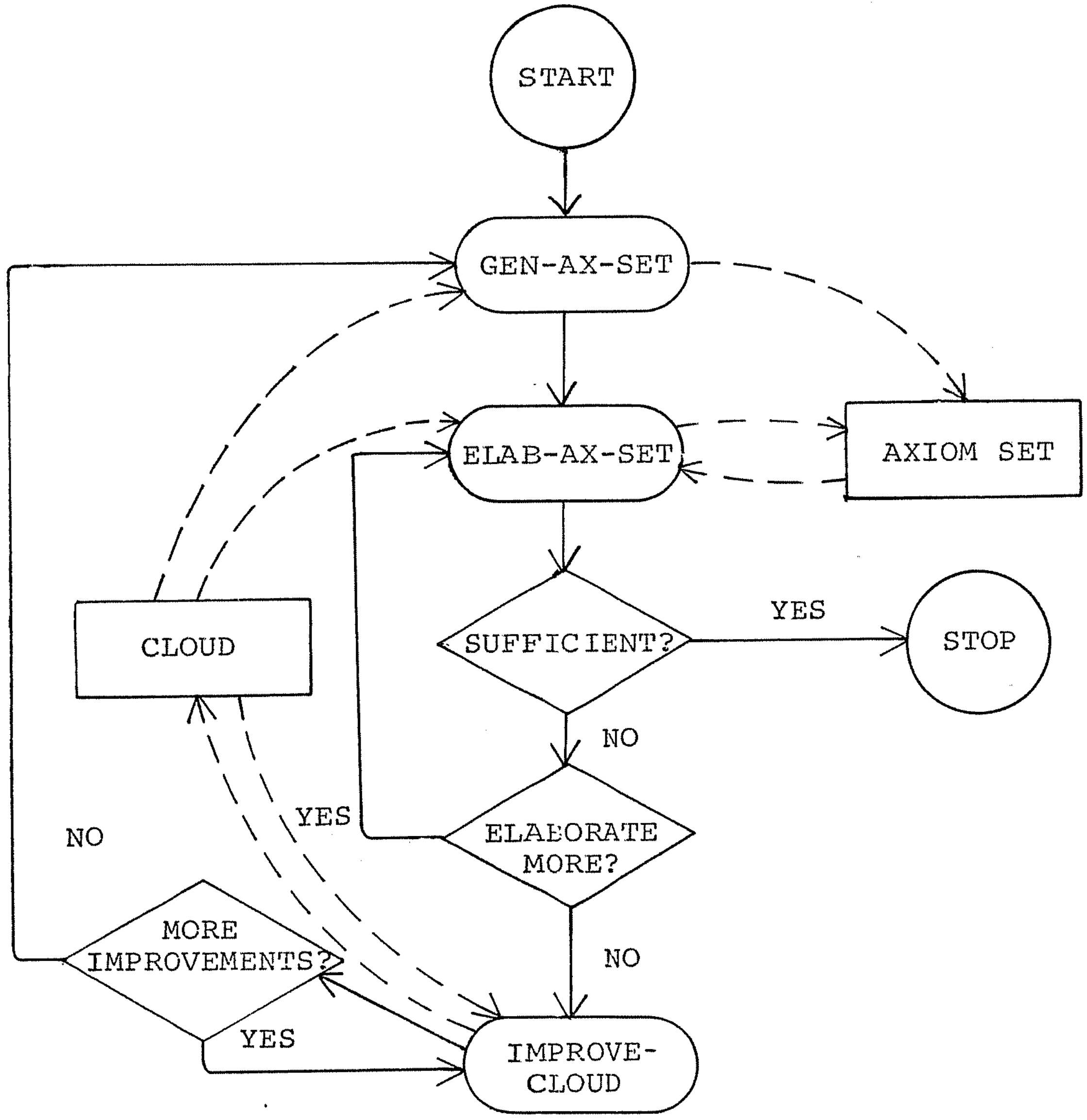


Figure 2: An example process to generate an axiom set from a cloud. The solid lines show control flow; the dashed lines show data flow.

also be used to develop a measure of error and a standard of comparison for increased automation of testing.

2.2.2 Requirements Phase

The starting point for a requirements process must inevitably be some informal, incomplete, and possibly inconsistent desired behavior for a system. The requirements process must produce a well-defined, complete, consistent, unambiguous, and testable set of system requirements which ensure that system behavior will be satisfactory to the customer and that the required system can plausibly be designed and implemented. The resulting requirements specification must also be suitable for the remainder of the developmental process.

The BMD system does not exist in a vacuum. Its whole purpose in being is to monitor the external environment and, under certain circumstances (the detection of threat missiles), to take actions to alter it (to fire and maneuver interceptors to destroy the incoming missiles). The environment thus not only affects the BMD system, but also is affected by it, so that a kind of feedback loop exists. Thus in order to test the desired behavior of the system, there must be a closed way of characterizing the physical world system as well as the BMD system, and also the coupling between them.

The specification of a closed system includes the "environment" with which all interactions take place. An open system may include an interface specification to the environment but does not include that environment. An open system is not testable without a specification of an environmental driver, and premature partitioning of a system design to provide such interfaces can introduce serious and artificial problems.

The requirements specification must be testable. This may be done either by specifying the system, interface, and driver or by simply specifying the closed system that includes the environment. Both options must be possible. However, we can maximize the scope of our formal techniques if the formal requirements approximation is the closed system. The partitioning of the closed system into system, interface, and driver could be done subsequently in a formal way if desired. Such a partitioning may not always be feasible or desirable, particularly at the initial informal level. The closed system specification is a complete and consistent specification whereas the open system, interface, and driver specification (informally arrived at) need be neither complete nor consistent -- and undetectably so in the worst case. We will, therefore, start our formal specification as that of a closed system.

Requirements elaboration can thus take place jointly in the environment and system models with a homogeneous technique. We can "design" and test the detailed environment at the same time and in the same manner as the system interacting with the environment.

The axiomatic model of the world (formally or informally specified by the previous axiomatic phase) can be approximated by an effective model of the world containing two systems, the physical world system and the (BMD) system being designed, with specified interactions linking the two. In our example, the BMD system would send (to the physical world system) radar signals and receive radar returns, and also send interceptor commands based on its interpretation of the radar returns. In this model, the interceptor process in the physical world system would be a function of the interceptor's previous position and velocity, and of the interceptor command, which is based on the BMD system's imperfect knowledge of the missile's position and velocity (derived from the interpretation of the radar returns). The difference between the effective specification (based on imperfect knowledge) and the axiomatic specification (based on perfect knowledge) can be taken as a measure of the error in the system.

Now we can specify some general procedures which can be used to build a process which creates the effective specification of the physical world system and the BMD system.

GEN-SYS: CLOUD x AXIOM-SET --> PHYS-WORLD SYS x BMD-SYS

This procedure takes the axiomatic specification and the current cloud as input, and generates the effective specification of the physical world system and the BMD system consistent with the input and with real-world laws.

ELAB-BMD: BMD-SYS --> BMD-SYS

This procedure elaborates the specification of the BMD system using decomposition and integration (both formal and informal), partitioning, etc., to produce a new specification of the BMD system.

FEASIBLE-BMD: BMD-SYS x BMD-SYS --> BMD-SYS

This procedure receives as input a current specification of a BMD system and a proposed new specification. The output of the procedure is the new (proposed) specification if it is feasible, and the original one otherwise. Determining feasibility may involve developing a design and elaborating different aspects of it to arbitrary depths. Figure 3 shows an example process that can be built using these procedures.

The difference between the initial and final requirements specification state lies only in the level of approximation to the axiomatic specifications and in the level of elaboration required for testability. The form of the specification does not change. Indeed, with "perfect" designers the initial specification might also be the last. Requirements specification will thus consist of an axiom set and an effective model containing a physical world system and a control system.

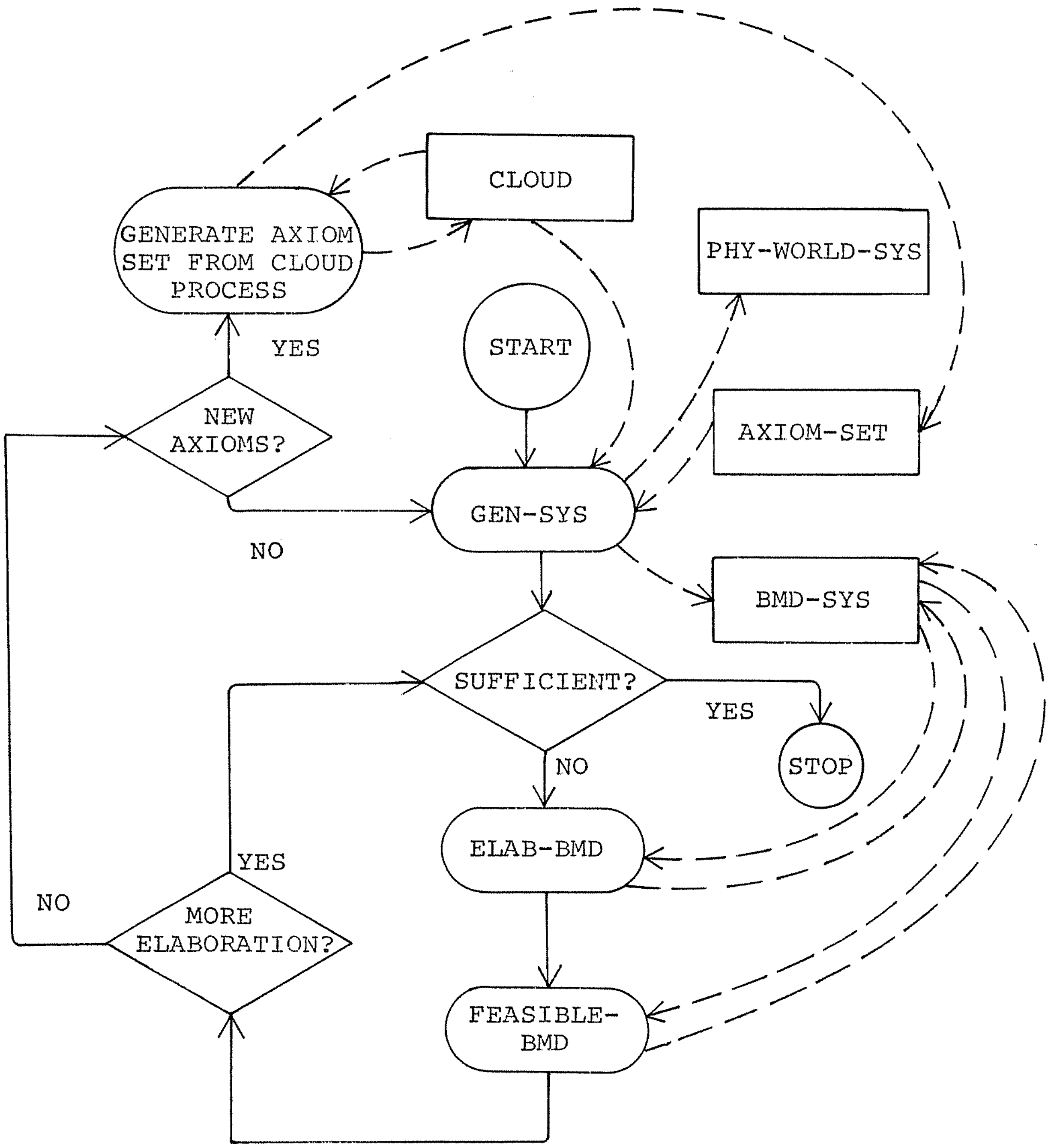


Figure 3: An example process for creating the effective specification of the physical world and BMD systems.

2.3 Decomposition/Integration

2.3.1 Requirements Decomposition

At any phase of the development process, it may be feasible to factor the development process into relatively independent processes that are subsequently integrated. There are two basic forms of such factorization which we call decomposition and partitioning. Partitioning techniques will be discussed in the next section.

A system may be decomposed into a set of its abstractions, each abstraction being a specification of a subset of the system attributes and values. The entire set thus completely specifies the system. Each member of the set completely specifies an abstraction of the entire system and is, therefore, testable for requirements. Two such abstractions may have overlapping attributes, with the design responsibility either shared or delegated to one of them.

Each decomposed system abstraction now initializes a development process of its own with possibly interacting steps. At some stage the set of elaborated specifications of each system abstraction must be integrated into a single system specification.

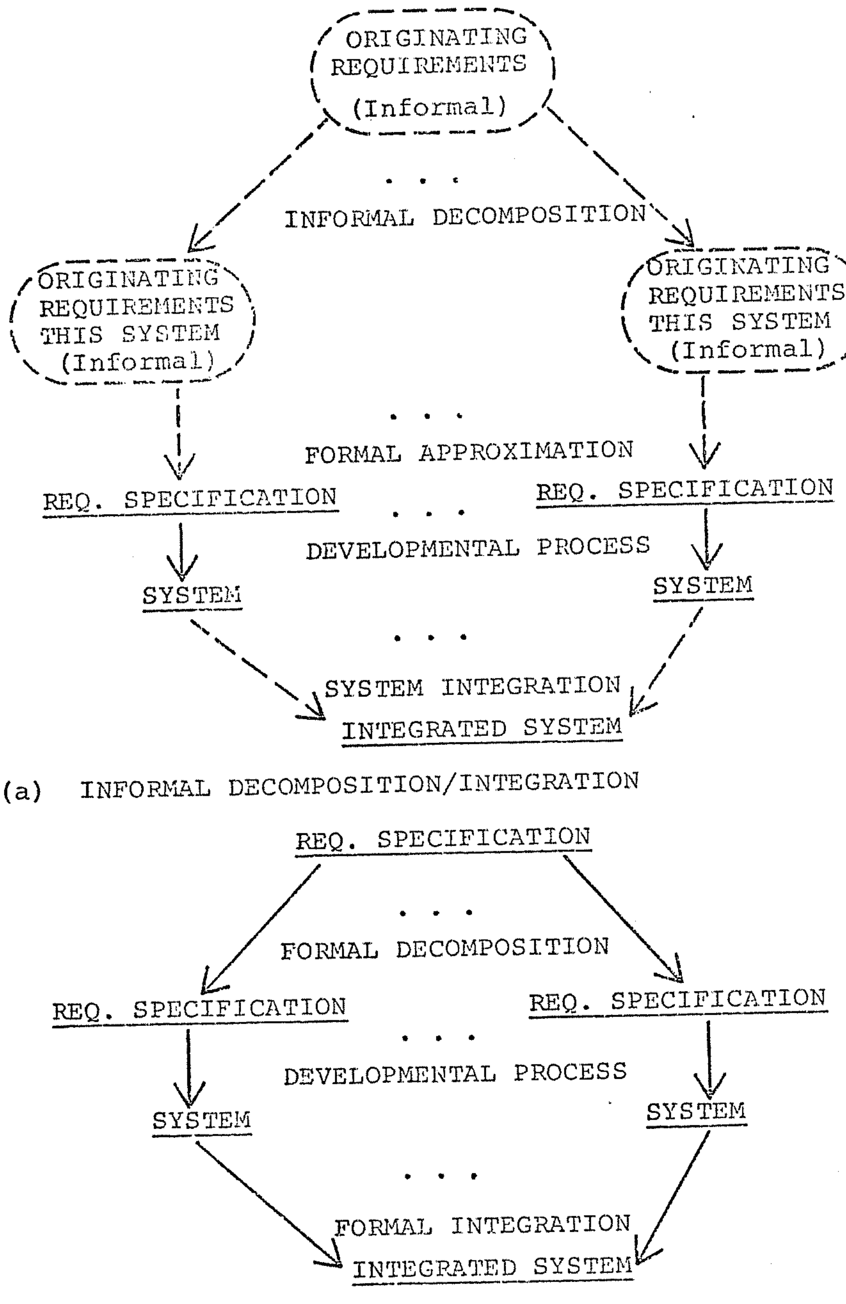
2.3.2 Informal Decompostion /Integration

A large system development will have many attributes that are only very loosely coupled, for example, the physical site design and the data processing design. Design decisions local to different design processes are not sensitively coupled and, within predesignated limits, may be made quite independently. Thus some forms of decomposition can be carried out at the earliest stages, thus factoring the development process into independent processes until the corresponding system integration occurs.

The earliest possible point of decomposition is in the informal originating requirements as shown in Figure 4a. We can decompose into closed systems by identifying tightly coupled subsets of attributes that are only loosely coupled with other subsets. Then by partitioning the attributes we can specify a closed system approximation for each attribute subset. The decomposition of a closed system is thus into a set of closed systems, each representing the original system from a different viewpoint.

The number of such decomposed systems is clearly application-dependent and sensitive to the identification of loosely coupled subsets of attributes. Some, such as physical site systems, logistic systems, logical systems, etc. are clearly loosely coupled since, within predetermined limits, design decisions are essentially independent. Those limits can be built into the requirements for each decomposed system. The application of our development process may now be made independently (and tested independently as well).

There may be some originating requirements that can not be analyzed for attribute coupling that prevent such



(a) INFORMAL DECOMPOSITION/INTEGRATION

(b) FORMAL DECOMPOSITION/INTEGRATION

FIG. 4: System decomposition/integration. Solid lines represent formally defined entities. Dashed lines represent informally defined entities.

decompositions. Since the originating requirements are not usually unique expressions, perhaps one can find a better set of requirements which can be decomposed. In any case, some non-decomposable requirements may still remain. These requirements thus can only be tested on the resulting integrated system. If such a requirement is tightly coupled to attributes scattered across the decomposed system, then the decomposition itself is probably ill-advised since system integration may produce expensive test failures. Perhaps one can find another equivalent set of originating requirements that can be more completely decomposed.

We can minimize the risk caused by non-decomposable requirements by carrying the separate development processes only far enough so that integration at that level (perhaps a different level for each system, or even within each system) makes the non-decomposable requirement testable. We must then deal with multi-level abstract system integration and testing in our requirements process.

Another serious problem arises from the informal nature of these decompositions. The subsequent integration must also be informal. Since there is no formal characterization of how they were taken apart, we are unlikely to formalize how they are put back together. In any event only ad hoc techniques can be used. Note that if a formal decomposition can be made as in Figure 4b, this problem can be avoided by doing it that way.

2.3.3 Formal Decomposition/Integration

A formal requirements specification can be decomposed in the same way as informal decomposition, except that the decomposition can be formally characterized and the possibility of subsequent formal integration can be tested _a priori_. The essential difference between formal and informal decomposition lies in the formal specification of the system being decomposed in the former case. We can thus precisely characterize the decomposition (even if we have no effective procedure for carrying it out) and establish sufficient conditions to ensure the correctness of the decomposition. Both the decomposition and the integration may be substantially aided by design automation tools. Indeed, we can accept sufficient design laws on the form of the decomposition to ensure that integration can be done and tested automatically. We should still support integration at many levels of design detail to minimize risk from non-decomposable requirements.

An example of a possible data processing system decomposition and integration is given in Figure 5.

In this case we can develop translators and translator writing systems to aid the integration of the decomposed systems.

This type of decomposition may be very powerful in isolating the effects of changes to one of the decomposed systems. Similarly, because of the loose coupling between such systems (or they would not be decomposed), some of the

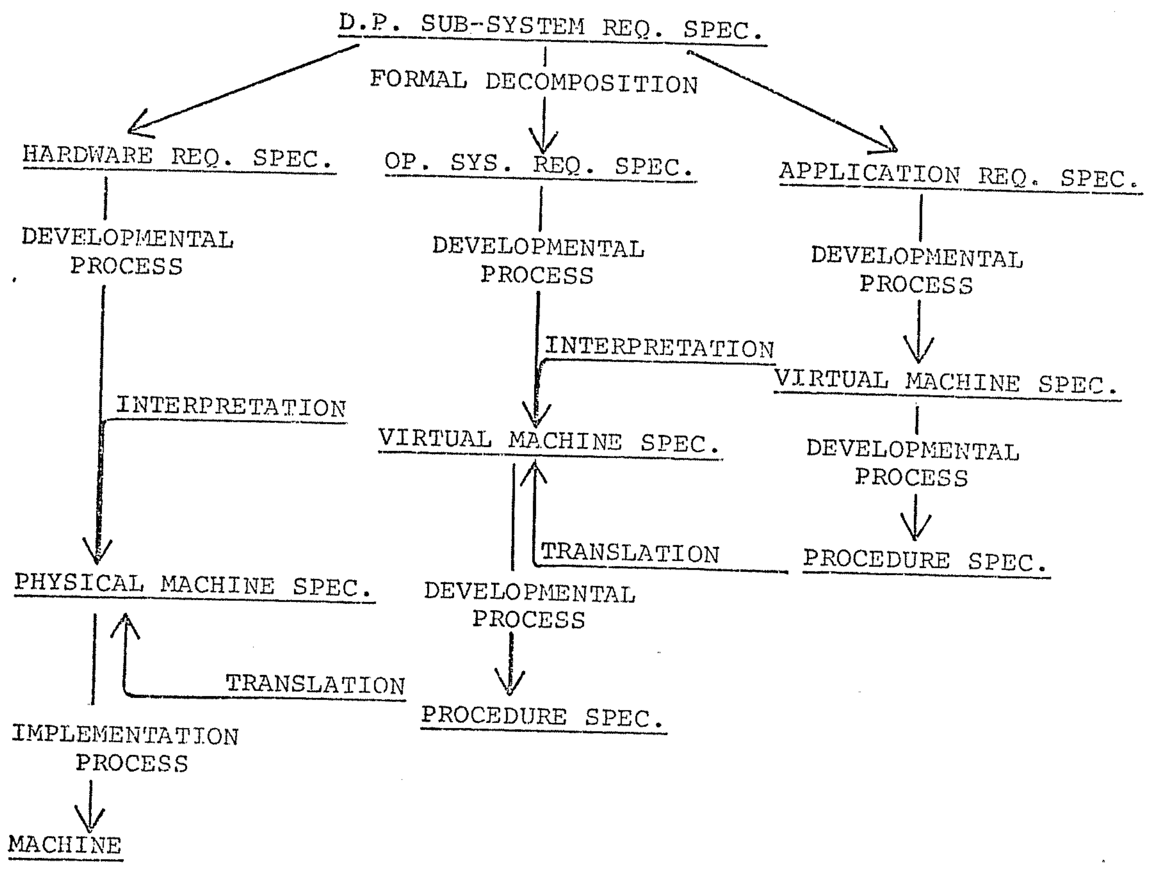


FIG. 5: An example of formal system decomposition/integration. The effective property of each formal specification ensures that the specified system is in a form interpretable by a formally universal interpreter (the simulating system). Translation (compilation) steps may improve efficiency of the testing of resulting implementation. The example developmental process does not exhaust all possibilities but is intended as an illustration.

components may be directly usable in other applications, or easily adaptable to changes in an application.

Because of the critical nature of some performance requirements and the extreme difficulty of meeting them, there must be some "escape mechanism" that allows application system designers to require direct hardware implementation of some application algorithms. Thus some need for interactions between development processes may exist and should be supported. Such interactions can be well-defined steps of the development process.

2.4 Partitioning

2.4.1 Requirements Partitioning

So far we have not looked at the required internal structure of a single, formal system requirement specification. We have discussed and justified the use of closed system specifications on the basis of testability and closure. Thus we implicitly assume there exist at least two interacting systems to be specified, the environment and the environment-manipulating system. The requirements process may, of course, require elaboration of both systems to reach an acceptable level of testability. Our formal specifications of a system must be able to require a set of interacting systems with a formal specification, not only of the systems, but also of their interactions. At any phase of the development process it may be desireable to partition the logical processes into closely coupled clusters, or nodes. The remaining steps of the development processes can then independently (or nearly so) elaborate the design of each node, maintaining the inter-node interactions as design invariancies. The partitioning of logical functions among the nodes does not necessarily imply the same partitioning of physical systems in the implementation. This point is illustrated in Figure 6.

2.4.2 Partitioned Testing

The logical partitioning of Figure 6 involves only a part of the formal requirements. For our purposes, we can factor system requirements into the following:

- Performance: how well must the system work?
- Resources: what kind and how many can the system use?
- Logical: what functions must the system support?

We are disregarding here the valid need for similar requirements on the requirements process itself, and we are not formalizing the testing of those requirements. This is a potentially important exception, and the need for better management tools for the developmental process is recognizable. We feel that formalizing the development process and the testing of developing systems are essential first steps in solving management problems.

The testability of a partitioned logical requirement specification is still dependent on the existence of the other logical specifications since only then do we have a

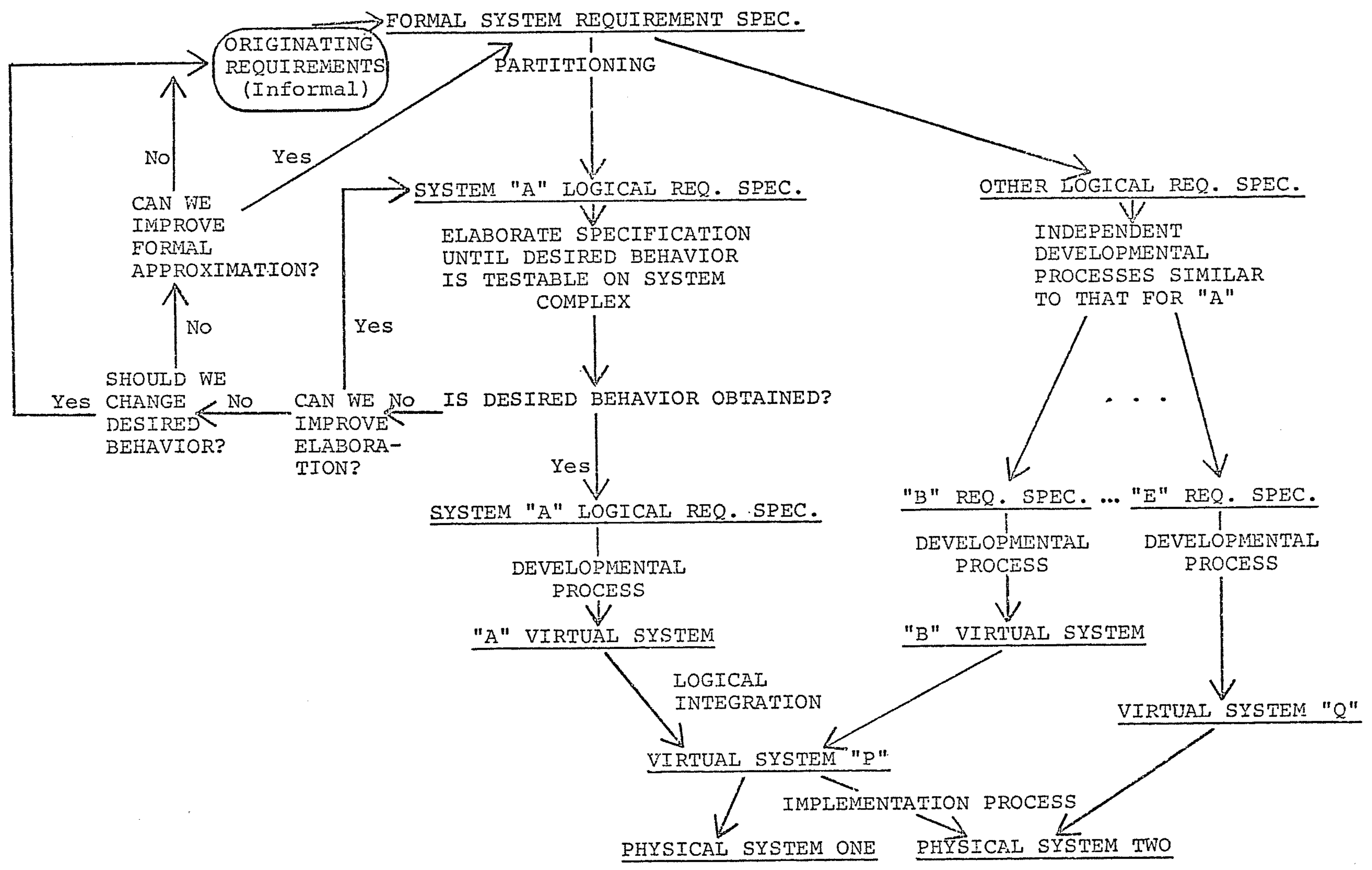

FORMAL SYSTEM REQUIREMENT SPEC.
ORIGINATING REQUIREMENTS (Informal)
PARTITIONING
No
Yes
CAN WE IMPROVE FORMAL APPROXIMATION?
No
SHOULD WE CHANGE DESIRED BEHAVIOR?
Yes
No
CAN WE IMPROVE ELABORATION?
No
Yes
SYSTEM "A" LOGICAL REQ. SPEC.
ELABORATE SPECIFICATION UNTIL DESIRED BEHAVIOR IS TESTABLE ON SYSTEM COMPLEX
IS DESIRED BEHAVIOR OBTAINED?
Yes
SYSTEM "A" LOGICAL REQ. SPEC.
DEVELOPMENTAL PROCESS
"A" VIRTUAL SYSTEM
LOGICAL INTEGRATION
OTHER LOGICAL REQ. SPEC.
INDEPENDENT DEVELOPMENTAL PROCESSES SIMILAR TO THAT FOR "A"
. . .
"B" REQ. SPEC. ... "E" REQ. SPEC.
DEVELOPMENTAL PROCESS
DEVELOPMENTAL PROCESS
"B" VIRTUAL SYSTEM
VIRTUAL SYSTEM "Q"
VIRTUAL SYSTEM "P"
IMPLEMENTATION PROCESS
PHYSICAL SYSTEM ONE
PHYSICAL SYSTEM TWO

FIG. 6: Logical requirements partitioning

testable closure. We can use for this purpose the currently most suitable of the other specifications in the partitioned set. If the performance and resource requirements are loosely coupled, we could use formal decomposition instead, and make the testing even more factored and localized as in Figure 4, but that would not be a partitioning step.

We must thus assume that performance and resource requirements are not practically decomposable into relatively independent requirements for each system in the partitioned set. Any attempted decomposition would run into "the requirements allocation problem" which in these terms is insoluble. Thus we won't try to solve it, but we don't need to in order to carry out the requirements process of Figure 6.

2.4.3 Parametric Partitioning

We may be able to do even better if we can develop parametric logical specifications for each member of the partitioned set. We can then use the decomposition and integration steps of Figure 7. All testing can now be done independently within parametric ranges.

Each of the decomposed systems is a system complex representing the entire system and is thus a testable entity. Interactions between the decomposed requirements processes are now required only when parameter ranges must be exceeded to meet specifications, and when the partitioned

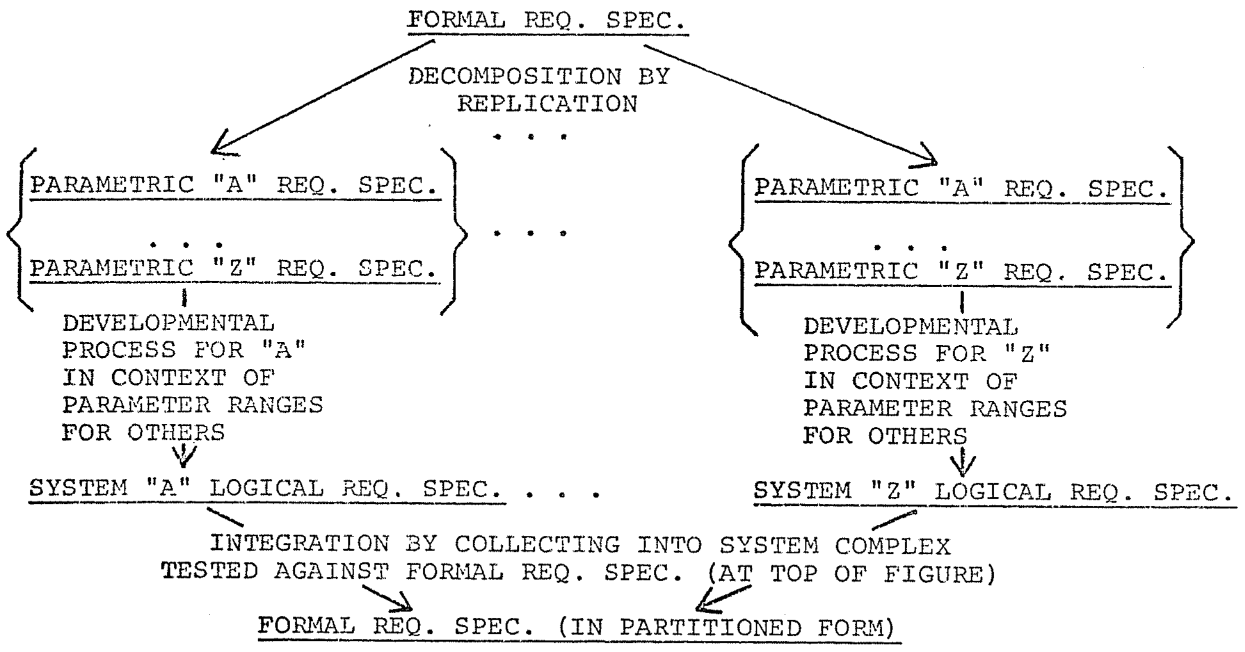

FORMAL REQ. SPEC.
DECOMPOSITION BY REPLICATION
PARAMETRIC "A" REQ. SPEC.
PARAMETRIC "Z" REQ. SPEC.
DEVELOPMENTAL PROCESS FOR "A" IN CONTEXT OF PARAMETER RANGES FOR OTHERS
DEVELOPMENTAL PROCESS FOR "Z" IN CONTEXT OF PARAMETER RANGES FOR OTHERS
SYSTEM "A" LOGICAL REQ. SPEC.
SYSTEM "Z" LOGICAL REQ. SPEC.
INTEGRATION BY COLLECTING INTO SYSTEM COMPLEX TESTED AGAINST FORMAL REQ. SPEC. (AT TOP OF FIGURE)
FORMAL REQ. SPEC. (IN PARTITIONED FORM)

Figure 7. Parametric partitioning.

systems are integrated by collecting them into a system complex.

2.5 <u>Primitive Elaboration</u>

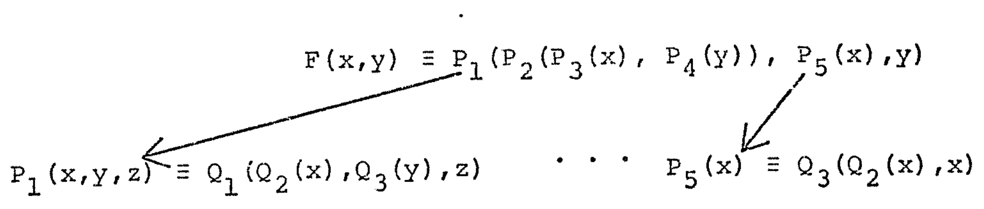
2.5.1 Functional Specifications

We will assume that the formal requirements specification will be defined using a functional formalism designed to provide a sufficiently general model for all systems of interest. This does not constrain the level of the specifications. There previously was no such model available, primarily because of the need to functionally model asynchronous interactions which was not met in available models. We have developed such a model and it is described in section 3. In this section we will ignore the requirements for formal interactions, although after section 3 they can be dealt with without change in the discussion presented here.

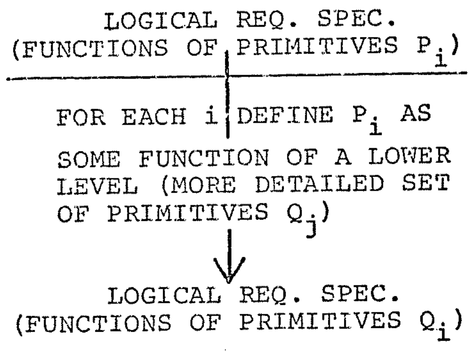
The initial formal requirements specifications will use rather high-level primitive functions in precisely specifying the system. Primitive functions are axiomatically defined or are informally characterized (e.g., in English descriptions). Each such initial primitive may eventually be elaborated by the development process into many processes or procedures spread over an entire network of implemented (physical) systems. The initial definition is thus as some mathematical expression of the high-level primitives.

2.5.2 Level of Detail

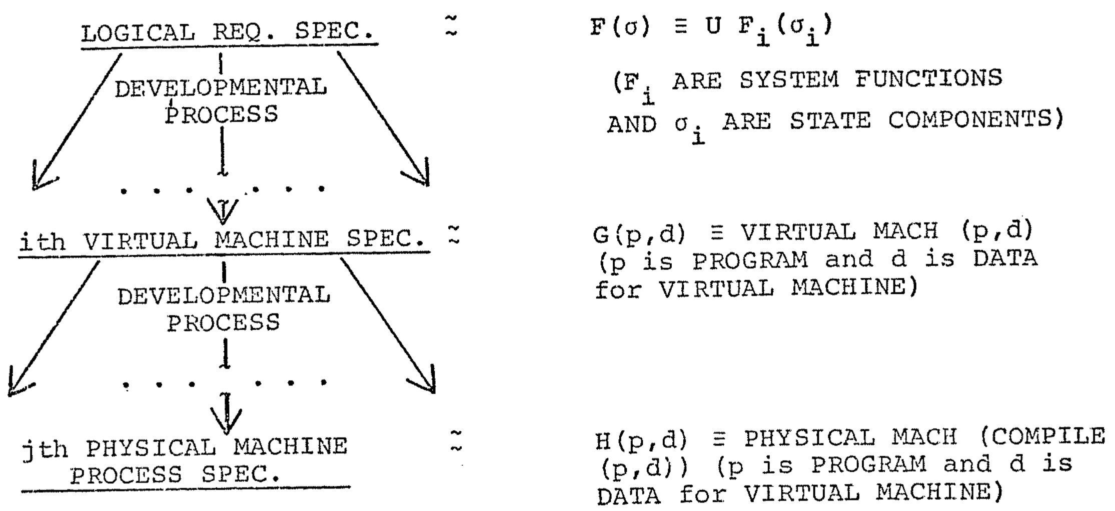
The level of formally defined detail (primitives are only informally defined) may not be sufficient to formally encode all of the originating requirements or to formally test against the originating requirements. The required elaboration of detail is obtained, as shown by Figure 8a, by formally defining the high-level primitives in terms of lower level ones, thus formally encoding in the defining expressions at least part of what was previously encoded


F(x,y) ≡ P1(P2(P3(x), P4(y)), P5(x),y)
P1(x,y,z) ≡ Q1(Q2(x),Q3(y),z)
. . .
P5(x) ≡ Q3(Q2(x),x)

(a) Function Definition Tree


LOGICAL REQ. SPEC.
(FUNCTIONS OF PRIMITIVES Pi)
FOR EACH i DEFINE Pi AS SOME FUNCTION OF A LOWER LEVEL (MORE DETAILED SET OF PRIMITIVES Qj)
LOGICAL REQ. SPEC.
(FUNCTIONS OF PRIMITIVES Qi)

(b) Primitive Elaboration Step


LOGICAL REQ. SPEC.
≈
DEVELOPMENTAL PROCESS
F(σ) ≡ U Fi(σi)
(Fi ARE SYSTEM FUNCTIONS AND σi ARE STATE COMPONENTS)
ith VIRTUAL MACHINE SPEC.
≈
DEVELOPMENTAL PROCESS
G(p,d) ≡ VIRTUAL MACH (p,d) (p is PROGRAM and d is DATA for VIRTUAL MACHINE)
jth PHYSICAL MACHINE PROCESS SPEC.
≈
H(p,d) ≡ PHYSICAL MACH (COMPILE (p,d)) (p is PROGRAM and d is DATA for VIRTUAL MACHINE)

(c) Levels of Primitive Elaboration

FIG. 8: Elaboration of a Functional Specification

informally in English. This results in an elaboration step as shown in Figure 8b.

The degree of such elaboration increases as we move along the development process. The requirements process ends when all requirements have been formally encoded and tested for suitability. Figure 8c describes some possible intermediate states in the development process. The virtual systems are used (as discussed in section 5) to factor the development process and are defined by some interpretation function (processor) operating on a pair of program and data (system state) to define a computation of the virtual (not physical, but logical) machine. Eventually the virtual machine programs and data may be compiled to implementation (physical) machine initializations. This design process is further described in section 6.

2.6 Requirements Process Summary

The informal originating requirements must be encoded formally in some functional specification whose behavior approximates that of the desired system. The behavior must be testable and, if unsatisfactory, either the originating requirements or the formal functional specification must be changed to improve the degree of approximation. The level of detail formally encoded may need to be elaborated prior to testing. We have described a model for both problem and requirements development. The requirements process ends when the current originating requirements are formally encoded and the specified system has satisfactory behavior. This may have required partial completion of the remainder of the development process.

We have identified several types of requirements process steps (e.g., approximation, decomposition, integration, partitioning and elaboration) and discussed the issues involved in their formalization.

We have studied the issues of formal testability and have described an approach to their resolution by the formal, functional, effective specification of the requirements for closed systems.

We have identified a number of important properties a formal system specification must have in general, and laid a foundation for the subsequent work in section 3. Other required properties will be developed, after the formalism is established, as design laws which ensure that the tools and tests discussed in this section can actually be provided. Further design laws will be derived from studies described by the remaining sections.

3. FUNCTIONAL PROCESS SPECIFICATIONS

3.1 Introduction

Given some informal requirements for a system, we wish to encode them in a formal specification. We must be able to verify that a formal specification corresponds to required behavior. For this verification we must be able to observe the formal specification's behavior, which entails the ability to observe the well-defined states and interactions of the specified systems. We may think of these well-defined actions as state transitions of a digital process, and the well-defined interactions as interface transitions. The state transitions can be defined as algorithms (possibly nondeterministic) for the successor state.

Many required systems have behavior which is naturally factored into the behaviors of several components, particularly for geographically distributed processing. However, these components are required to communicate and coordinate behavior via some form of interactions--the state of one component has an effect on the state of the other component. It is also important that these interactions may occur asynchronously, as the indispensibility of interrupts has shown. We must have a formal model for such interacting systems.

3.1.1 Basis for Functional Specification

From the goals for modelling system behavior and for observing and verifying specified behavior, we have the concepts of state, algorithmic state transition, and interacting component processes of a system. Our formalism begins with these concepts and develops accordingly the goals and properties required for a specification formalism.

What should a functional process specification look like? There is a generally accepted concensus that a process can be defined by a set Σ of process states and a (possibly nondeterministic) successor function f. The application of f to a process state σ to produce a successor process state σ' is known as a process step. When we wish to specify a more complex process in which internally asynchronous or independent transitions occur, there is no longer a consensus, and more work is needed. Ramamoorthy and So [Ram76] have said that a functional process specification should be "(1) comprehensible, (2) unambiguous, (3) verifiable, and (4) machine processable". These goals are certainly justified by the need for a requirement methodology: (1), (2), and (3) are needed for correctly and consistently formalizing functional requirements, (2) and (3) are needed for correctly developing and implementing specifications, and (4) is needed for accuracy, for the

[Ram76] Ramamoorthy, C.V. and So, H.H., "Survey of Principles and Techniques of Software Requirements and Specifications", Department of Electrical Engineering and Computer Sciences, University of California, Berkeley, 1976.

volume of the work, and for simulation testing. These goals may be met by a formalism based on a mathematical notation of functions and sets which has been subject to constraints which guarantee (3) and (4) and which allow sufficient expressive power for (1).

A further goal of formal requirement specifications is to ensure properties which we desire the specified system transitions to have, such as: algorithmic implication (computations terminate without blocking), the ability to test real-time systems by non-real-time simulation, and the ability to model any characteristics of a real system by our specifications (completeness from 2.1.3.3). In general, a specification should be such that each of a set of relevant properties is either guaranteed by the form of the specification, or is efficiently decidable from the specification. Furthermore, a specification should be efficient in the sense that we should be able to limit the information in a specification to that necessary to ensure these properties or their efficient decidability. We would also like our specification formalism to allow expansion of this goal as useful new properties are discovered and included in the relevant set (for example, properties relating to the evolutionary development of existing systems).

Use of a functional notation has many nice properties. Specifications based on mathematical function notation allow concentrating on relevant areas of a system and hiding the rest within primitive functions; this is the high level property mentioned in section 2.1.3.3. Such specifications also permit guaranteeing properties or their easy decidability by axiomatic constraints on function combinations, using much of what is already known about function behavior. Simple primitives for interactions may be inserted into the formalism of mathematical functions quite naturally, and these interaction primitives may also be handled by axiomatic constraints. Such a formalism also ensures the consistency and unambiguity of a specification if only minimal care is taken.

3.1.2 Properties of a Specification

In justifying the mathematical form of a functional process specification we have made reference to desirable properties for a specfication as well as to more general goals in the development of functional specifications. The properties include:

(1) observable and verifiable behavior of a specification.

(2) generality for asynchronously interacting processes.

(3) algorithmic implication (all state transitions will complete).

(4) testability -- particularly of real-time distributed processes by simulation.

(5) completeness of specification with respect to characteristics of required system.

(6) ability to superimpose development and evolutionary processes on the specification formalism.

(7) ability to concentrate only on areas of the system relevant to desired analysis, leaving low-level details within primitives.

(8) consistency and unambiguity of specification.

In addition to these properties there are two mentioned in section 2.1.3.3:

(9) effective decidability of behavior of a system specification.

(1Ø) traceability of the impact of changes in a system specification.

The decidability of behavior must be in terms of analyzing asynchronous interactions, which will be discussed later. Tracing the impact of changes depends upon the change being local to an area of the specification, which in turn depends upon a correct design decision in factoring the specification.

3.2 System Specifications

We start our discussion of formal specifications by introducing some basic definitions. (A more rigorous and complete treatment is given in Appendix A.) We also discuss graphical representation of processes.

Definition. A value space V is a set of values v which are not here further defined.

Definition. A state component σ_i is a subset of a value space V_i.

Definition. A state component space Σ_i is the power set of V_i.

Definition. A state space Σ is a product

$\Sigma_1 \times \Sigma_2 \times \ldots \times \Sigma_m$ of state component spaces, i.e. $\sigma \in \Sigma$ if and only if $\sigma = (\sigma_1, \sigma_2, \ldots, \sigma_m)$ where $\sigma_i \in \Sigma_i$, $i = 1,2,\ldots,m$.

Definition. A process is a pair (Σ,f) where Σ is a state space and f is a possible nondeterministic state successor function. The state successor function f may possibly be decomposed into component successor functions f_i where the f_i are set functions. A component successor function f_i may possibly be further decomposed into value successor functions f_{ij} where the f_{ij} are not set valued functions but are value space valued functions. Note that either the f_i or the f_{ij} may be nondeterministic. The definitions of the function decompositions are developed in the next section.

A computation of a process (Σ,f) is a sequence $\sigma^0, \sigma^1, \sigma^2, \ldots, \sigma^i, \ldots$ such that $\sigma^i \in \Sigma$ $(i \geq 0)$, and σ^0 is an initial state of the computation. Thus a process and an initial state define a computation.

3.2.1 State and Interaction Graphs

Here we will consider two graphical representations of a process and argue for the superiority of one. Suppose that we have a successor relation f which we wish to decompose into simpler relations. Since only finite specifications of f are useful, writing a different relation for each single state will fail if there is an infinite number of states. The solution is to gather states into a finite number of equivalence classes and write a separate successor relation for each class.

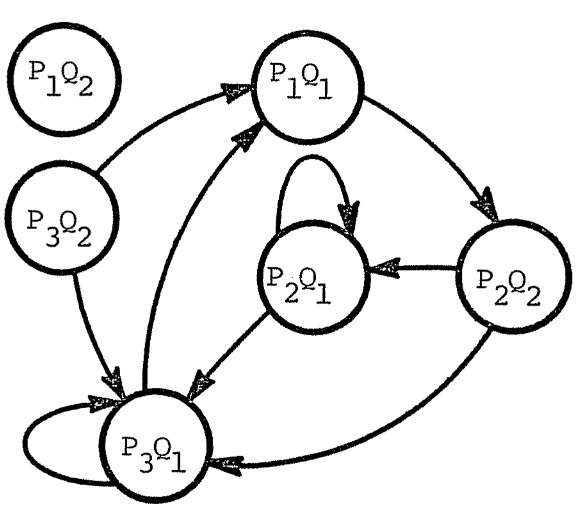
If the equivalence classes are represented as nodes of a graph and the successor relations as arcs, the result is known as a <u>state graph</u>. The corresponding state successor relation can then be defined as a finite state machine. State graphs may be useful for forming specifications, even for small systems, only as long as the designer has a single locus of control transitions. However, state graphs unfortunately have complexity problems. Consider a process which is the composition of two loosely coupled processes P and Q. As P cycles through m state equivalence classes and Q cycles through n state equivalence classes, the composite process will be cycling through a state graph of mn nodes. For each of the m possible values of component P, there will be a different variant of the component successor relation for Q. Also, we lose our conceptual picture of the separate subprocesses by forcing a coupling that does not exist.

A state of the composite process could be represented as having two state components, one giving the state of P and the other giving the state of Q. This indicates a better way to graph the composite process: the graph will basically be the union of the state graph for subprocesses P and Q. We will encode interactions between state transitions of P and Q via touching arcs. (See figure 9 for an example.) This kind of graph will be called an <u>interaction graph</u>.

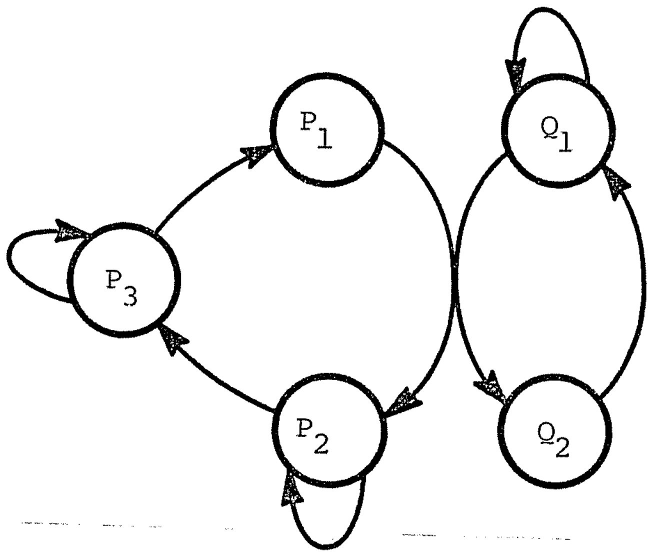
The interaction graph is a better characterization of what is going on in the composite process, and is much simpler, especially when m and n are large and P and Q are loosely coupled. The state graph explicitly encodes information that the interaction graph only implicitly encodes, such as the fact that P_1Q_2 and P_3Q_2 are unreachable in figure 9, for instance. However, this information is irrelevant to the problem of specifying f. If that information is relevant to some analysis, it may either be obtained by studying computations via the interaction graph or it could not have been obtained in the first place.

3.2.2 State Successor Function Decomposition and Process Graphs

The decomposition of a state successor function f into component successor functions can be represented by a <u>process graph</u>. Each node of the graph represents a state component space and each arc represents a component



(a) <u>state graph</u>



(b) <u>interaction graph</u>

Figure 9: State and interaction graphs of a process consisting of two loosely coupled subprocesses P and Q. The state space of P is partitioned into P_1, P_2, and P_3 while the state space of O is partitioned into Q_1 and Q_2. Intuitively, Q waits in state Q_1 until P reaches P_1, and then they can proceed independently back to the waiting states P_1 and Q_1.

successor function f_i. The arc is drawn from the component spaces in the domain of f_i to the component spaces in the range of f_i. For example, suppose we have the state successor function $f: \Sigma \rightarrow \Sigma$ where $\Sigma = \Sigma_1 \times \Sigma_2 \times \Sigma_3 \times \Sigma_4$. Suppose further that f can be decomposed into component successor functions f_1, f_2, f_3 given by:

f_1: $\Sigma_2 \times \Sigma_4 \to \Sigma_1 \times \Sigma_3$

f_2: $\Sigma_1 \times \Sigma_2 \to \Sigma_2$

f_3: $\Sigma_3 \to \Sigma_4$.

Then f can be represented by the process graph in figure 10.

3.3 Functional Notation

3.3.1 Primitive Functions

The f_i and f_{ij} in a functional specification may be left as primitives or may be decomposed into lower level primitives. The functional specification must be based on primitive functions of the designer's choosing. These primitive functions may be arbitrarily simple or arbitrarily complex. The more complex the primitives are, the simpler the resulting specifications structure will be, and the less help the designer will receive in analyzing it. The primitive functions must of course obey the design laws, in order to ensure the overall specification properties which the designer desires. The ability to select these primitives freely lets the designer avoid the formalism if he wishes. He may elect to define f as a primitive, in

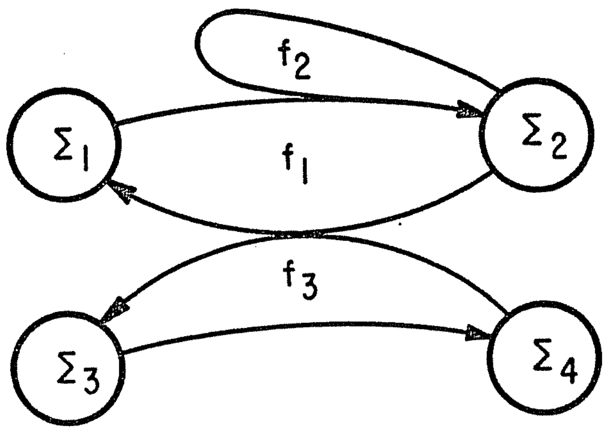


Figure 1∅: A process graph with component selector functions. Note that we just indicate domains and ranges.

which case there are no restrictions on it. Neither, of course, will he receive much help in analyzing it.

3.3.2 Operations on Functions

We must now decide what basic operations on functions we must have in creating the functional specification structure. Inspired by recursive function theory we will use a few primitive functions and the operations of composition, primitive recursion, and selection. The operation of function composition must be included; with it we decompose f into the f_i, the f_i into f_{ij} and high level primitives into lower level primitives. We include the operation of primitive recursion because it gives us the capability for bounded iteration. Finally we use the operation of function selection, which is defined as follows: select$(p_1:g_1,\ p_2:g_2,\ \ldots,\ p_{k-1}:g_{k-1},\underline{\text{true}}:g_k)$ evaluates to the value of the first g_i such that p_i evaluates to true (the p_i are predicates which evaluate to true or false). This selector function gives us a model of control.

It is worth noting that the basic form of a component function is that of a tree of nested functions with primitives as leaves. This form is established by the composition schema. Recursion and selection do not alter the tree form of the structure which is finally evaluated, but only delay its binding until evaluation time. Recursion finally expands to a fixed depth nesting, and selection simply reduces to the selected subtree.

At this point we can model any primitive recursive state successor function for a single system. We now need to introduce a functional model for interactions to deal with multiple interacting systems.

3.4 Interaction Specifications

3.4.1 Exchange Functions

We now define a class of primitive functions which will allow the designer to specify interactions. These *exchange* functions have the unique property that under certain conditions they will exchange values of arguments with a matching exchange function elsewhere in the specification. The exchange of arguments between a pair of matching exchange functions is accomplished by having each of them evaluate to the argument of the other. Exchange functions are labelled with subscripts and only exchange functions with the same label can match. The set of exchange functions with a given subscript is referred to as a class.

The three exchange functions XC, XA, XS are defined as follows:

$XC_i(\alpha) = \beta$ if there is an outstanding $XC_i(\beta)$ or $XA_i(\beta)$ which has been waiting for a matching exchange function,

or

if this $XC_i(\alpha)$ has been waiting for a matching exchange function and an $XC_i(\beta)$, $XA_i(\beta)$, or $XS_i(\beta)$ is evaluated.

$XA_i(\alpha) = \beta$ if there is an outstanding $XC_i(\beta)$ which has been waiting for a matching exchange function to be evaluated,

or

if this $XA_i(\alpha)$ has been waiting for a matching exchange function and an $XC_i(\beta)$ or $XS_i(\beta)$ is evaluated.

$XS_i(\alpha) = \beta$ if there is an outstanding $XC_i(\beta)$ or $XA_i(\beta)$ which has been waiting for a matching exchange function to be evaluated, and

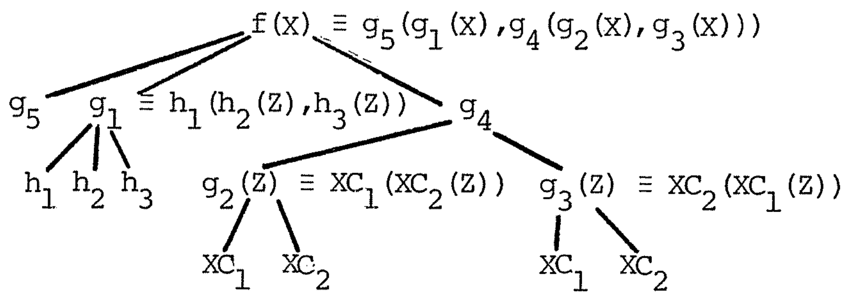
$= \alpha$ otherwise.

3.4.2 Evaluation

Any state successor function can be defined by a definition tree as shown in Figure 11a and automatically transformed into a corresponding precedence graph as shown in Figure 11b. The precedence graph simply displays the constraints on possible evaluation sequences. The use of exchange functions imposes additional (and potentially incompatible) synchronization constraints and allows values to be exchanged.

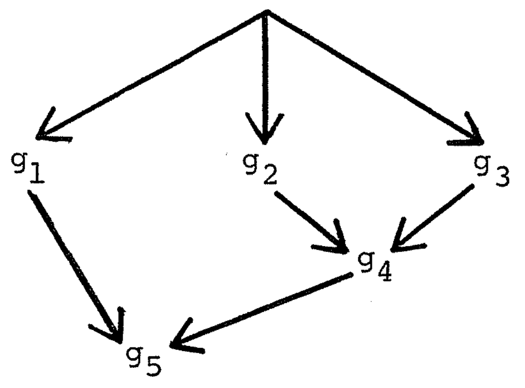
The exchange functions can be analyzed as normal (possibly nondeterministic) functions in their local context while still providing a high level (non-procedural) model for asynchronous interactions in process specifications of internally and externally asynchronous processes.

An internally asynchronous interaction could be defined as matching exchanges between component successor functions. An externally asynchronous interaction could be defined as matching exchanges between state successor functions (each defining an independent system). Thus all interface interactions are modelled directly and homogeneously by our functional specifications.

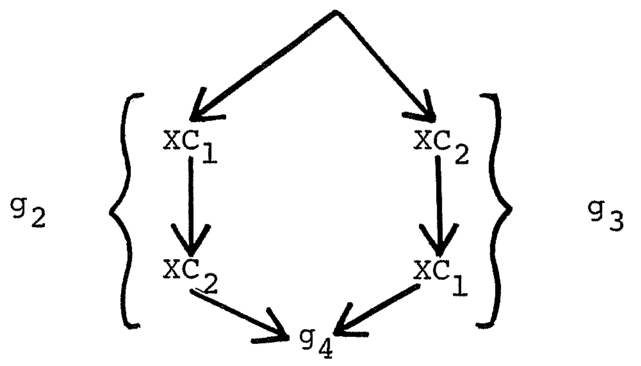


(a) A definition tree for $f(X) \equiv g_5(g_1(X), g_4(g_2(X), g_3(X)))$.

The functions $g_4, g_5, h_1, h_2, h_3, XC_1, XC_2$ are primitives.



(b) The precedence graph for $f(X)$ in terms of g_i.



(c) A blocked precedence graph for $g_4(XC_1(XC_2(A)), XC_2(XC_1(B)))$; control cannot pass the first XC_1, XC_2 functions.

Figure 11: Use of exchanges in a function.

Using an immediate exchange, XS, we can also model what we will call unsynchronized systems as containing only XS type inter-system interactions. Such systems, which never wait on any interactions, are essential for many real-time systems and for modelling the environmental system or real, physical world. An XS function cannot be allowed in an intra-system interaction since there cannot be sufficient constraints in a precendence graph to ever force its instantaneous matching with another exchange (creating blockage problems). Such use cannot be allowed.

As illustrated above, the enormous generality of functional interaction specification comes at the price of some new design laws governing the use of exchanges. Arbitrary usage can lead to inter- or intra-system deadlocks (as must be true for any general interaction model). An example of an intra-system deadlock is given in Figure 11c when no other exchanges of those classes are present. However it is possible to place restrictions on the form of the specification such that no process will be blocked in this way. Each restriction will correspond to a design law which must be followed in order to guarantee completion. Such restrictions are given in Appendix B.

3.4.3 Simple Examples

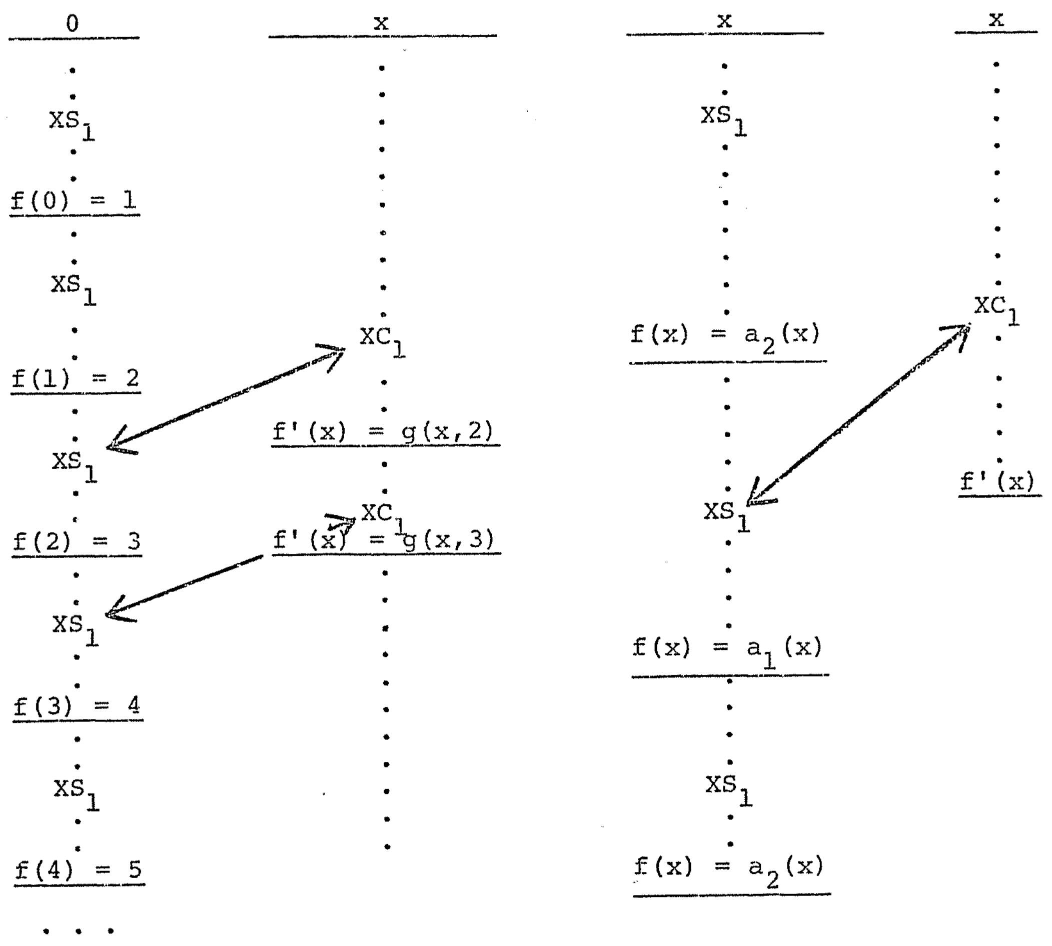
A few trivial examples may help explain the use of exchanges.

We could define a pair of interacting systems by state successor functions f and f' as given in Figure 12a. In this example the evaluation of f' is delayed until the exchange XC_1 has been completed by a subsequent evaluation of XS_1 in f. Thus the evaluation of f is not so constrained, since XS_1 will exchange with itself in order to continue without delay. f' could thus be interpreted as a system synchronized to the system f, that uses values from f in its own computations. The system f could be interpreted as a simple real-time clock that goes on with its cycling (ticking) without delays or synchronizations with another system. A sketch of the computations of f and f' is given in Figure 12a.

As a second example, f and f' can be defined as in Figure 12b. In this example f' is synchronized as before. The system f can now be consisdered to cycle through evaluations of $a_2(x)$ unless an XC_1 is outstanding. In that case, the value of XS_1 will be T and the function $a_1(x)$ will be evaluated instead. The system f could thus be described as having been interrupted by system f', to perform function a_1.

3.4.4 A Spooling Example

We present here a functional specification of a spooling system. The spooling system will be divided into a set of asynchronous processes, each with its own state space and successor function. From our intuitive knowledge of



(a) Real time clock example

$f{:}z \to z \quad f(x) \equiv \mathrm{First}(\mathrm{suc}(x), xs_1(x))$

$f'{:}z \to z \quad f'(x) \equiv g(x, xc_1(0))$

Where first $(x,y) = x$ and

$\mathrm{suc}(x) = x+1$ and

z is the set of integers.

(b) Interrupt system

$f{:}z \to z \quad f(x) \equiv (XS_1(F){:}a_1(x), T{:}a_2(a))$

$f'{:}z \to z \quad f'(x) \equiv g(x, xc_1(T))$

Where T ≡ True and F ≡ False

and z is the set of integers.

FIG. 12 : Simple Exchange Examples

what spooling system means we know that the following subsystems can be and should be asynchronous (therefore minimizing implementation constraints).

a.) input from real world model (RWI)

b.) card reader (CR)

c.) input queue (IQ)

d.) output queue (OQ)

e.) line printer (LP)

f.) output to real world (RWO)

We include a model of the external world so that no external interaction is required.

System Specification:

The states of all systems are vectors of elements from the value space V, which is the set of all job records. Each vector represents a bounded queue, where the rightmost element is the head of the queue and the leftmost non-null element is its tail. The null element in V will be denoted by $\emptyset$. Let Σ_V be the state component space of the value space V. Let Σ_V^m be $(\Sigma_V^{(1)} \times \dots \Sigma_V^{(m)})$. Let $\sigma_{O,X}$ be the initial state of process X. Then we have the following state spaces and initial states:

$\Sigma_{RWI} = \Sigma_V^m \qquad \sigma_{O,RWI} = (\sigma_1, \dots, \sigma_m)$

$\Sigma_{CR} = \Sigma_{IQ} = \Sigma_{OQ} = \Sigma_{LP} = \Sigma_V^4$

$\sigma_{O,CR} = \sigma_{O,IQ} = \sigma_{O,OQ} = \sigma_{O,LP} = (\emptyset,\emptyset,\emptyset,\emptyset)$

$\Sigma_{RWO} = \Sigma_V^m \qquad \sigma_{O,RWO} = (\emptyset,\emptyset,\emptyset,\emptyset)$

The σ_i's represent all the jobs that move through the spooling system in a day. Each job starts in the queue

$\sigma_{O,RWI}$ and is passed along the pipeline until it ends up in the state of RWO. The intermediate queues of the pipeline must be bounded to allow us to specify this system. Here the bound has been arbitrarily set at four.

Jobs are passed along the pipeline according to the following scheme: for every step taken by a process, it produces one element and passes it to the next by means of an XA. Also in every step it executes four XS's to receive elements from the preceding system in the pipeline. This allows the queues to grow and shrink independently, within the size bound of four.

The following state successor functions use these primitive functions and macros :

p_i = the projection function which returns the value of the ith argument.

card-to-data = convert card image to data image.

interpret = process some data.

data-to-print = convert data to print.

in-val $(v,\sigma) = \sigma$ if v is the null value, otherwise in-val returns a new vector formed by inserting the value v in the rightmost null position of the vector σ (i.e., add v to the tail of queue σ).

is-null (σ) = TRUE, if the leftmost element of the vector σ is null (i.e., there is room for another value at the tail of the queue σ); = FALSE, otherwise.

rs(σ) = the vector σ of values right shifted one place, with the vacated leftmost place filled in with the null element (i.e., the element at the head of the queue is removed).

H($1,$2) = SELECT (is-null ($1):

in-val($XS_{\$2}$(O),$1); true: $1)

H($1,$2) is a macro of two arguments defining a function which takes a vector of four elements and returns an updated vector. The updated vector is formed by taking the result of an XS and placing it in the leftmost null position of the original vector. If the original vector has no room, or if the XS returns a null result, then the function returns the original vector. In other words H updates the queue if possible by adding a new element at the tail of the queue.

G($1,$2) = H(H(H(H(rs($1),$2),$2),$2),$2)

G($1,$2) is a macro defining a function which takes a queue and updates it by removing the element at the head of the queue and filling in the queue with the results of as many XS's as yield non-null results. $1 is the queue argument and $2 is the class of the XS function.

RWI successor function:

$f(\sigma) = p_2(\text{SELECT}(\sigma_m \neq \emptyset : XA_a(\sigma_m), \underline{\text{true}} : \emptyset), rs(\sigma))$

CR successor function:

$f(\sigma) = p_2(\text{SELECT}(\sigma_4 \neq \emptyset : XA_b(\text{card-to-data}(\sigma_4)),$
$\underline{\text{true}} : \emptyset), G(\sigma,a))$

IQ successor function:

$f(\sigma) = p_2(\text{SELECT}(\sigma_4 \neq \emptyset : XA_c(\text{interpret}(\sigma_4)),$
$\underline{\text{true}} : \emptyset), G(\sigma,c)$

OQ successor function:

$$f(\sigma) = p_2(\text{SELECT}(\sigma_4 \neq \emptyset : XA_d(\text{data-to-print}(\sigma_4)), \underline{\text{true}} : \emptyset), G(\sigma,c))$$

LP successor function:

$$f(\sigma) = p_2(\text{SELECT}(\sigma_4 \neq \emptyset : XA_e(\sigma_4, \underline{\text{true}}: \emptyset), G(\sigma,d))$$

RWO successor function:

$$f(\sigma) = \text{in-val}(XC_e(\emptyset),\sigma)$$

3.5 System Complex Specifications

3.5.1 System Specification Domain

Our specification restricts the class of solutions and implementations we will accept for the system complex. In particular, the type of functional process specifications we have defined requires that there are observable closed states which processes and their components must pass through. These states, which are needed for verification and testing, are generated by the constraints on events (function evaluation) which bind the synchrony of these events. However we may defer decisions about synchrony or asynchrony until their proper context by including the relevant events within a single process step, or if necessary by placing the events in different processes. Thus we may generate observable closed states at the levels where they are needed and defer other synchronizations. The necessity for placing events in different processes may arise from an unsynchronized components system (e.g., the physical world) or when there is no functional requirement for the relative rates of occurrences of those events, or no such requirement in the present design context. If relative rate requirements exist we may be able to include the events within one process, where desired properties are more easily ensured or decided. The flexibility which our functional specifications provide with respect to observed closed states and binding of synchrony is itself a desirable property.

Our process specification formalism is capable of specifying a large range of systems in a natural way. Its constructs reflect the ways we think about the systems under consideration, subject to the derived constraints which guarantee desirable properties of system behavior. Our functional framework lends itself to analysis of axiomatic constraints, yet within that framework we express the breaking of a state into components, the distinction between operations on sets and operations on values, and the decomposition of functions by the operators necessary for primitive recursive functions (composition, selection, and primitive recursion). Furthermore within the functional framework we express three primitive types of interaction, XS, XA, and XC. With these exchange functions we can express a wide variety of interactions. It is also important that we can express a wide variety of constraints on these exchange functions in order to ensure the non-blocking properties of asynchronous interactions; it is not unreasonable to say that asynchronous interactions create many of the problems in large distributed real-time systems. There is no generally accepted model of all asynchronous interactions, as there is of recursive functions, so it is difficult to claim that our exchange functions are completely general. However, they are adequate to model asynchronous and real-time interactions of existing computer systems.

3.5.2 Simulation and Testing

Good functional specifications allow simulation testing which is consistent and complete with respect to the behavior of the specified system. Our functional specifications can automatically be interpreted as a simulation model of the specified systems. Our formalism allows the stochastic simulation of both functional evaluations and relative occurrence times of events even without formal specification of primitive functions. A complete procedural model may also be generated when procedures for primitive functions are supplied. Simulating a functional specification is relatively easy, and can be done formally. The direct use of the functional specifications as the testable model prevents erroneous assumptions or obsolete models from invalidating the testing.

A very useful property, but one which can be ensured only by correct decisions in stating requirements, is the local testability of other properties of the specification. That is, we wish to be able to ensure or decide desirable properties by considering small portions of the total system specification. Finding axiomatic constraints which have an effect on this local testability would be an important area of future work.

3.6 Summary

The formalism for functional process specifications must allow analyses for the properties needed by the requirements methodology. Furthermore, these properties are motivated by the specific types of systems required, that is real-time distributed data-processing systems. Some of these properties are listed in sections 2.1.2 and 3.1.2. Others are developed in following sections. In the case of certain properties, for example algorithmic implication and boundedness, we have indicated how the analysis of a functional specification may be carried out. Certain properties will be satisfied only if correct decisions are made in developing specifications; these include the locality of testing and the traceability of the impact of changes on the specification. Some of the properties are guaranteed by the definition of the formalism, for example generality and the ability to use high level primitives.

The high level (informally specified) primitives are especially important for the design process methodology, as discussed in section 2.5. High level primitives make it possible to factor requirements testing into well-defined, manageable steps. Primitive elaboration also constitutes a well-defined design step, and allows design decisions to be made in a local context.

The formalism we have defined allows the formal development of a requirements methodology for real-time distributed data-processing system. We have begun an analysis of properties of the formalism and have indicated areas needing further analysis. Although more design laws will be imposed, the functional formalism developed here seems a very suitable basis for identifying, studying, and solving the remaining problems.

We can now give useful formal functional specifications of networks (interacting complexes) of real-time systems and their real world environment. This can be done for any stage of the development process from requirements to implementation. Further, we may use the same functional formalism to study and define developmental process themselves.

4. REAL-TIME PROCESSES

We now have a formal way to define interacting processes. Thus we can address the questions of what a real-time process is, and how it can be tested.

4.1 Definitions: the following are some working definitions we will use.

A *free-running* process is one which has no interprocess XC or XA interactions. It is a process which, by its very nature, cannot wait for other processes, i.e., it *does* not wait. It will continue to run as the real-time clock runs: the missiles will continue their flight even if the data-processing system is deadlocked; the radar signals will move through the air regardless of who is listening. These processes will have only XS interactions.

A process which has an XC/XA interaction with another process is *synchronized* with that process. Such a process cannot be independently clocked or measured. If both processes have XC/XA interactions, they are *mutually* synchronized.

A *cluster* of processes consists of two or more processes that interact with each other.

The *rate* of a system is the time for the execution of one system step. The rate need not, and often will not, be constant from step to step. We will also use the rate of the process to mean the rate of the system which implements that process.

4.2 Characterization of a Real-Time Process Cluster

4.2.1 Definition

A *real-time* process cluster is one which must produce a correct response to a stimulus within a given amount of time [PhB76] [Hec76].

The designer of the process cluster must specify the stimulus, the response, and the time. The stimulus will presumably be the completion of a synchronizing interaction with an inherently free-running process. (If the process were not inherently free-running, one could simply have it synchronized with the appropriate process, and the need for the response "in a given amount of time" would be eliminated.) The response will be the execution of any function designated by the designer. This could be the occurrence of an interaction sending control information to the free-running process, or the decision as to whether or not to execute that interaction, or being ready to receive another stimulus, etc. The time bound may be absolute

[PhB76] Phillips, Jorge V., and Bredt, Thomas H., "Design and Verification of Real-Time Systems", Proceedings of the 2nd International Software Engineering Conference, San Francisco, California (October 1976), pp. 124.

[Hec76] Hecht, H., "Fault-Tolerant Software for Real-Time Applications," ACM Computing Surveys, December 1976, pp. 391-408.

(e.g., 100 milliseconds) or non-absolute (e.g., 100 milliseconds at least 80% of the time). See section 4.2.6.

There are three critical aspects to the response:

1. It must be a "correct" response.
2. It must be logically deliverable in time.
3. It must be physically deliverable in time.

The first requirement, that of the response being "correct", is not unique to real-time process clusters. This idea will be developed more fully later. The second requirement is necessary but not sufficient for the third. The first and second are properties of the functional specification; the third is a property of its implementation.

Thus it is the notion of a time bound (logical and physical) that seems to differentiate real-time from non-real-time process clusters. Yet it is not the existence of a time bound that makes a cluster real-time; it is the <u>requirement</u> of a time bound for proper functioning of the cluster.

The proof that a required time bound will be met must be followed from the functional specification of the application process cluster through the virtual and physical operating systems and finally to the hardware system which implements the application process cluster. The properties needed to prove that the time bound will be achieved must be preserved as the process cluster is translated through these levels.

Numerous studies comfirm the fact that the later a design error is found and corrected, the more costly it is [DrK76]. Because of this, testing should be done at the earliest stage feasible. Thus, although some testing must be delayed until the implementation stage, the logical specification of the process cluster rather than the implementation should be testable for behavior. Because of the importance of early testing, in this section we will in general concentrate on proving properties of the application process cluster.

4.2.2 Interactions with Free-Running Processes

In many real-time process clusters, the response to a stimulus from a free-running process is an interaction which affects the future states of that process. For example, the responses might be to change the setting of some variable in a nuclear reactor, or to maneuver an interceptor aimed at an incoming missile. The time bounding is important because these processes are free-running and therefore <u>do</u> <u>not</u> <u>wait</u>. If the response takes too long, it may be too late to keep the reactor from exploding or the missile from reaching its target. In general, the response will have an effect on this free-running process by changing its otherwise predetermined course of action, and this change will affect the next stimulus this process initiates. Thus a kind of feedback loop is established.

As another, more specific example, consider a free-running process containing the representations of the airplanes in a given air space, interacting with a process cluster which functions as the air traffic controller. The airplanes will broadcast to the controller their position, altitude, speed, and direction. They will continue to fly and thus change the values of these variables according to a predetermined course. However, a message from the controller to change altitude, speed, etc., will result in (probably) different changes in value, and these will become new stimuli for the controller.

Because the free-running process will continue to run, not only the content but also the timing of the response is important. For example, the commands from the controller to two potentially colliding airplanes to climb and dive respectively are useless if they arrive after the two planes have met in midair. Thus the time bound can be due to the very nature of the non-synchronizing way the free-running process interacts with the process cluster. An on-line interactive airline reservations systems would not be considered real-time by this definition, but the air-traffic controller would be.

Alternatively, a time bound could be required even if the response would not directly affect the free-running process. For example, in a real-time monitoring process the desired response might be the logging and analysis of a stimulus, and getting ready to receive the next stimulus.

While the time between stimuli received would have no effect on the free-running process being monitored, the designer might decide that if the times were more than 100 milliseconds apart, the monitor would not reflect the monitored process as accurately as necessary. Then he could impose a time-bound requirement.

4.2.3 Logical and Physical Time Bounds

Note that the time-bound requirement is both logical and physical. The existence of a bound is a logical characteristic of the logical processes; the actual time bound is a physical characteristic of the implementation. Thus a process cluster could be shown to be logically correct and bounded, but could have a physical time bound which would be in practice physically impossible to achieve. Thus the logical bounding is necessary (if not bounded, there is no minimum evaluation speed sufficient to meet requirements) but not sufficient. In this section we will concentrate on the logical time bound, but it should be remembered that implementation contraints may later force redesign of a system which is logically correct.

4.2.4 Virtual Implementation

Let us distinguish between two levels of the implementation which we will call the virtual implementation and the physical implementation. The virtual implementation has a unlimited supply of processors and of space.

Therefore competition for these resources does not exist. Performance could be predicted based solely on the number of times each primitive would be executed (either as worst case or based upon a distribution function, perhaps with correlations to other distribution functions), and on the time required for the execution of those primitives in the given implementation. The maximally parallel execution of these primitives would essentially give us the best possible time for a given implementation, with no interference because of competition for resources. The specification can then be tested for behavior in the virtual implementation. If the virtual implementation is not fast enough to produce the desired behavior, then either a new specification or a new implementation must be employed, since the best case of this specification's implementation is not adequate.

4.2.5 Physical Implementation

If the desired behavior is produced in the virtual implementation, then the physical implementation must be analyzed. It is here that the loading, i.e., the effect of competition which was abstracted out of the virtual implementation, is analyzed for its effect on the performance. While the analysis of the virtual implementation would require only the frequencies of the primitives used, the physical implementation analysis would also take into account the correlations between frequencies of primitives, as these correlations would affect the resource utilizations. The aim would be either to find the maximum loading at which the desired performance could be achieved for the specific physical implementation, or alternatively to find the physical implementation required so that a given loading would meet the performance requirements (i.e., the number of processors and the amount of space required so that the physical implementation would yield the same behavior as the virtual implementation). If neither of these is satisfactory, (e.g., if the physical implementation is fixed or too limited and the maximum loading permitted is expected to be exceeded), then if the loading rises above the maximal level at which the desired performance can be achieved, perhaps a different specification could be employed, where a degraded mode of operation is specified. This degraded specification would presumably have decreased frequencies and perhaps different correlations, resulting in a higher maximum loading permissible. (If the maximum loading is not higher, this specification should obviously be abandoned, as its performance would be no better and its behavior degraded from the original specification.) Perhaps several levels of "graceful degradation" would have to be specified to allow the performance requirements to be met by the expected levels of loading. Then an upgraded implementation (for example, with another processor or additional space) would presumably only increase the maximum loading at which the given specification would give the desired behavior.

However, the analysis might show no increase, indicating that the added power is not at the bottleneck, and thus that the money for such an "improvement" could be more profitably spent elsewhere.

4.2.6 Absolute vs. Non-Absolute Time Bounds

We have been discussing a time bound here as though it were an absolute time bound. It is true that one would often prefer the time bound to be absolute. Yet one could alternatively specify the probability (or distribution function) that a given time bound will be reached. This could give the designer considerably more freedom than an absolute bound, since he would not have to solve the worst case. For example, worst-case allocation of resources may not be required; a very good heuristic may be used in place of an expensive algorithm. Yet the additional flexibility also imposes certain constraints:

1) The probabilities must be figured accurately and reliably.

2) One must be able to know when the time bound has been exceeded.

3) One must specify appropriate recovery action to be taken if the time bound has been exceeded. The "recovery" could be simply notifying someone of the situation, or it could be an intricate procedure to continue as best as possible. (See [Hec76] for a similar approach.)

There may be certain times when a bound must be absolute: it must be attained 100% of the time. For example, one may be willing to pay any price necessary to keep a nuclear reactor from blowing up: expensive algorithms rather than efficient heuristics, extra hardware so that worst-case allocation is possible -- even abandoning the project if the money or current technology is not sufficient to provide a guarantee. Yet often to make any design practical or even technically feasible, one must accept a certain amount of risk. For example, one may be forced to accept a certain probability of "leakage" of offensive missiles that will not be intercepted.

4.2.7 Defining Time Bounds

A relevant way to logically express the time bound required for the response to reach the free-running process is in terms of the number of steps of the free-running process. For example, the response may be required within n steps from the stimulus. (If the minimum time for the execution of a process step of the free-running process is known, this could also be expressed as time. Section 4.4.1 discusses some of the problems of dealing with rates.) For the sake of simplicity, assume that only one process computes the response, and say that this process takes at most m of its process steps to do so. (Section 4.3 addresses some problems of proving this bound of m steps.) Then the ratio of the rates of the synchronized process to the free-running process must be at least m/n, i.e., the synchronized process must take at least m steps to each n steps that the free-running process takes. If the free-running process could wait, it could simply synchronize and wait for the other process to complete its m steps. However, if it must be free-running and therefore unsynchronizable, then the rate-ratio bound must be imposed.

4.3 Path Bounds

No rate can be proved to be fast enough to meet a time bound if the path from the stimulus to the response cannot be proved to be bounded. To do so, one must first of all be able to define that path, and then to prove that the path is bounded. Obviously, the complexity of the path will determine the ease with which this can be done. The following classification of kinds of paths is in order of increasing complexity.

4.3.1 Intraprocess Intrastep Paths

If the response to the stimulus is produced in the same process step in which the stimulus was received, then we know that the response path is bounded, if the state successor function is bounded. (Note that the path could be bounded even if the state successor function, while algorithmic, is not bounded, but the proof of boundedness would require additional analysis.) Our formal functional specification makes it easy to decide if a function is bounded, since building functions by using only the operations of composition, primitive recursion, and bounded subtree selection will intrinsically give a bounded function. Our formalism provides sufficient (though not necessary) conditions for boundedness; if interprocess exchanges are not used the formalism can thus be considered as a design tool for proof of boundedness on this level. The use of interprocess exchanges would require additional analysis.

4.3.2 Intraprocess Interstep Paths

Interstep but intraprocess computations, however, are not intrinsically bounded or even algorithmic; these properties must be proved to exist. A simple path without loops and without interactions would easily be proved to be bounded. However, the existence of loops would require an analysis for a bounding on the loop. Yet even the discovery of such a loop is not a trivial matter. For example, in bounded subtree selection, the first predicate could be used as a loop-exit condition, where its corresponding value could be the response or could trigger the response in some other function. Alternatively, the third predicate might indicate loop exit, in which case the negation of the first two predicates as well as affirmation of the third would be required for loop exit. These loop-exit conditions, as simple or as complex as they may be, could be used as induction variables. The value of the induction variable would cause the loop to be exited at a given point, namely when that variable takes on a certain value or range of values. If it can be proved that a traversal of the loop brings this variable closer to the exiting value, then by induction one can prove that the loop is bounded.

One may want to have design laws to control the complexity of the induction variable, as well as to make the induction variable easier to identify. If automatic discovery of the loop and its exiting conditions proves to be too difficult or too computationally complex, the designer may have to specify the loop and exit conditions, perhaps using automatic verification of the specified paths as a design tool.

4.3.3 Interprocess Paths

When the path involves interprocess interaction, more complexities are involved, for now at least part of the whole process cluster must be analyzed. For example, if our original process does an interprocess XC_i, it will wait until some other process does an interprocess XS_i, XC_i, or XA_i, unless an interprocess XC_i or XA_i is already outstanding. One must prove that if the latter is not true, then some other process will execute an interprocess XS_i, XC_i, or XA_i within a bounded amount of time. The levels may go deeper, of course: the interprocess XS_i from process A may not be executed until an XC_j is satisfied by an interaction from process B, etc. For the purposes of this interprocess interaction analysis, we can form precedence graph abstractions of the processes in which only the interprocess interactions are relevant. A design law requiring that intraprocess exchanges have different indices from interprocess exchanges would allow such an abstraction to be made. The designer could then work with these abstractions to determine if a given interaction would always complete, i.e., would be mated with another interaction, within a bounded amount of time. See Appendix B for further information on such mating theorems.

4.3.4 Others

We also have other options as designers. The case 2) specification may be given instead as a case 1), with initial to final state step occurring in only one state successor function evaluation. Some case 3) specifications may also be given with interactions between free-running processes as similar "single step" synchronized process specifications. Much more work will be required on this topic.

4.4 Attaining the Time Bounds

4.4.1 Specific Ratios vs. Lower Bounds on Ratios of Rates

Systems need to be robust with respect to their implementations, so that changing the rates of processors will not cause unforeseeable and unwanted behavior in the system. If the correct behavior of interacting processes is dependent on the ratio of the rates of the systems implementing them, a number of problems arise. First is the problem of defining what the rates are, especially since they can vary from time to time. The rates may be affected not only by the specific computations taking place but also by competition for resources and by the resource allocation algorithms which resolve the conflicts created by this competition. Thus if the correct behavior of unsynchronized processes relies on specific rate constraints, one must not only somehow define these ever-changing phenomena but also somehow achieve them. Even if this can be done, any perturbations in one system could radically change the behavior of the process cluster, which would then not be very stable or robust. However, if the processes are synchronized, then the implementer would only need to meet performance bounds. Any implementation that would exceed these bounds would also be acceptable, so that perturbations, as long as they left the performance above a minimum level, would not affect the desired behavior of the process cluster. A lower bound on this ratio of rates is obviously the only solution possible if the process cluster must endure and evolve over time; maintaining let alone achieving a specific ratio of rates would be highly impractical if not impossible.

4.4.2 Performance Graphs

What is needed is essentially a graph of the performance based on the rates of the systems involved. For the purposes of this analysis, the only relevant aspect of the performance is whether or not the behavior is acceptable; thus the line marking the boundary between acceptable and unacceptable behavior is what is important. Its intersection with the performance curve defines the permissable operating regions. Just knowing some characteristics of the boundary can be very useful, even if the exact location of that boundary is not known. For example, if acceptable behavior constitutes discontinuous points rather than a smooth surface, then finding appropriate rates for the systems will be difficult at best, and the behavior would be very unstable with respect to any perturbations in rates.

4.4.3 A Simple Example

As the simplest example possible, consider a single synchronized process which interacts with the free-running process. Assume that the rate of this free-running process is fixed, determined by physical factors in the real world (e.g., the force of gravity, or the speed of light). Now if the exact performance boundary is known, then the minimal rate at which the synchronized process must be run can be determined. This could result in the cheapest implementation which would give acceptable behavior.

However, suppose that the exact location of the performance boundary between acceptable and unacceptable behavior is not known, but it is known that the boundary is monotonically increasing with respect to the rate of the synchronized process. Then a working system could be upgraded (i.e., its rate increased), and the resulting faster system would also be a working system. This would mean that, provided a given minimal rate was achieved, the process cluster would be rate-independent. This would be expected with only a single synchronized process. Its only synchronizing interactions could be expected to be the receipt of the stimulus from and the sending of the response to the free-running process, as there are no other processes to interact with. (If there were an intermediate interaction, one could simply break the stimulus-response in two, where the new response to the original stimulus, and the new stimulus for the original response, would be this intermediate interaction). Thus for a given stimulus, the same response will always be produced in the same number of process steps. The rate at which the process is being run affects only the number of steps the free-running process will take between sending the stimulus and receipt of the response. The value and therefore the "correctness" of the response computed in the synchronized process is not affected by the rate of that synchronized process; only the timing is. Once the rate is sufficient so that the response is delivered "in time," any faster rates will also be "in time." Thus any single synchronized process could be considered rate-independent.

4.4.4 A Complex Example

Now consider the case where rather than a single synchronized process there is a synchronized process cluster. We can informally characterize the rate of this process cluster by the rate at which responses are given to the stimuli produced by the free-running system. As with the single process, the process cluster is rate-independent in that if a response is "in time" at a given rate, then it is also "in time" at a faster rate. However, unlike the single process case, the actual response may vary as the rates of the individual systems implementing the process cluster vary.

4.5 Constraints for Rate-Independence

4.5.1 Strict Rate-independence

What constraints must be followed so that a given process in the cluster would be rate-independent in the strictest sense of the word (i.e., increasing the rate of the system implementing the process would only increase -- or not affect at all -- the rate of the cluster)?

Constraint: All interprocess interactions must be predetermined with respect both to which process it will exchange with and to the message to be exchanged. This effectively eliminates the nondeterminism that can be associated with interactions. The only variable left is the amount of time that this process or the other would wait for the exchange to be paired. If the now faster process previously had to wait for the other process, it will simply have to wait longer, but the elapsed time will be the same, since the other process is the bottleneck. If the other process had to wait before, then it will not have to wait as long (or perhaps not at all) for the now faster process, so the elapsed time will be less, and so the rate of the process cluster will be increased.

What are conditions sufficient to meet the above constraint?

1.) The process must not send or receive any XS's. Obviously, the content of the XS messages could change as the rate increased.

2.) There can be only one possible candidate for matching an exchange. This preserves the determinism. This could be enforced by having only one other exchange with the same index, or by having all the other exchanges with the same index in the different branches of a subtree selector.

3.) Ensuring that the message to be exchanged is deterministic is more difficult. For instance, the process receiving a predetermined message from the potentially rate-independent process could use the time at which that message was received (which would vary with the rate) to decide what other message should be sent. Thus the second message would not be deterministic but rather would be determined by the rate of the process. To prevent this nondeterminism, the receiving process must also be rate-independent, i.e., have only XC exchanges with a single match and deterministic messages. This effectively means that the cluster cannot contain any free-running processes.

4.5.2 More General Pate-Independence

The above constraint would be easy to enforce, and does ensure the strictest kind of rate-independence, but it eliminates all interesting real-time process clusters. Let us relax the meaning of rate independence to mean that increasing the rate of some system in the complex will 1) not decrease the rate of the complex, and 2) the response will still be correct. This permits the nondeterminism which was disallowed previously. The price of the nondeterminism, however, is that all possible paths must be examined to ensure that they produce the desired behavior.

The nondeterminism of which exchanges would be matched could probably be analyzed reasonably efficiently if the designer is judicious in introducing such nondeterminism. However, analysis of the specific messages exchanged could lead to computationally explosive analysis of value correlations. We need to find design laws which are not overly restrictive but which allow efficient analysis to be done. Such design laws would eliminate worst-case process clusters but not most clusters of interest.

4.6 Non-Real-Time Testing

Simulation is one of the chief means of testing large complex process clusters. Since the actual timing is so important in real-time processes, they have traditionally been tested by real-time simulation. However, the real-time environment which must be simulated may get so complex that it can no longer be computed in real time. One would then want to do non-real-time testing, knowing that a successful test in a "slowed-down" real-time environment would imply a successful test in a real-time environment running at normal real-time speeds. What constraints on the processes are necessary so that such non-real-time testing will be valid?

First we must establish the level of the process being tested. Presumably prior analysis of the functional specifications affirmed that the stimulus-response path is bounded. Knowing that the path is deterministic and bounded independent of the relative speeds of systems already is very useful; it also means that real-time tests need only to look at path times and correlations. The simulation tests would be run on the actual implementation, since that is the level at which physical timing measures are meaningful. The difference between real-time and non-real-time testing is the speed at which the stimuli enter, i.e., the interarrival times. At the functional level this speed may not be relevant; it may have been abstracted out. If there are functional performance dependencies, they can of course be tested in a non-real-time way. However, at the actual physical implementation level, the interarrival times do affect the actual times for path traversal. Thus a virtual operating system with an infinite number of virtual resources would show no effect, but a physical operating system with a bounded number of resources would impose inter-performance dependencies and thus different timings.

The relevant factors for determining the effect of the timing include the number of virtual resources needed, the number of physical resources available, and the resource management policy which couples the two. Given these three factors, if one could prove that the rate of path traversal was not affected (or how much it was affected) by the number of paths being traversed (i.e., by the number of stimuli being processed at one time), then non-real-time testing should produce the same results as real-time testing. Perhaps analysis could also show the point at which non-real-time testing is no longer reliable for a given resource management policy (increasing the number of virtual resources required or decreasing the number of physical resources available could push a system past the threshold of reliable non-real-time testing). Such analyses would also give measures for evaluating different resource management policies. In the limit of light loading, physical resource contention has minimal effect. In this case the only real-time testing required is to find maximum loadings for given path performances. All other testing could be done non-real-time.

Note that this analysis relies on the proof that the rate of path traversal is not affected by the number of paths being traversed. Currently, there are no design laws which would make such a proof possible, nor even any strong indications that such design laws do exist in a form which is not overly-constraining. This is simply one possible approach to the problem.

Another approach involves taking measurements to determine path correlation. Thus one could measure the performance times on each path as a function of the stimulus frequency. Then one could determine the effect on performance when paths are coupled; this would show the effects of contention. From this data one could develop a model for the coupling between paths. This model could then be used to extend the results of non-real-time simulation testing to expectations of real-time testing results. Only real-time experiments to measure such correlations would be required.

4.7 Meeting Performance Requirements

4.7.1 Forecasting Performances Early

The only place one can really determine whether or not a given application process meets its performance requirements is in its final implementation, where it has been integrated into a virtual operating system, a physical operating system, and finally into the hardware. However, discovering so late in the design cycle that a given design will not meet the performance requirements means very costly and time-consuming redesign. One would like to be able to proceed with a design with some level of confidence that this design could meet performance requirements, and with design decisions being made so as to further enhance the probability of meeting the requirements. While the final determination could only be made at the end of the design cycle, a methodology which could provide some such confidence and criteria for design decisions could eliminate some (hopefully most) unacceptable designs early in their design cycle, thus saving the investment of time and money.

4.7.2 Forecasting at the Application Process Level

The forecasting of the eventual performance of the application process is based on the precedence graph of its functions. The time needed to traverse this graph of functions is the time for one process step. This time is determined by the longest path through the graph, which in turn is determined by the time needed to execute each function.

A crude, worst-case approximation of this time for a function's execution would be the time needed for each primitive's execution, multiplied by the number of executions (either expected or mean value, or maximum) of each primitive required in the execution of that function. This would be the time for serial executions of all primitives, and parallel executions, as we shall discuss later, can improve this total time. Unless these relative execution times of the various primitives are known, however, a vector of the frequency of primitives must be kept, with one frequency value for each primitive. For example, if a function required three executions of primitive A, no executions of B, and ten of C, its frequency vector would be (3, 0, 10) assuming that only these three are the only primitives available. The total frequency for all primitives can be useful only if the frequency of each primitive is weighted by its relative execution time before being summed. For example, if primitive A required 20 units of time, primitive B 50 units, and primitive C 30 units, then the frequency vector (3, 0, 10) could be replaced by the single value 20 X 3 + 30 X 10 = 360. However, since these values are not known at this time, the entire frequency vector must be kept.

However, a simple vector may not contain enough information, because of alternative branches which may be selected in bounded subtree selection. Each branch could be analyzed independently, resulting in a tree of vectors resembling the specification tree rather than a simple vector. Perhaps knowledge or hypotheses about the probability of taking each branch could be used to weight the frequency vector so that a weighted average could be used to approximate the tree of vectors. Either the expected value or maximum value of the iteration variable of a primitive recursive specification must be used, introducing another source of variation.

Regardless of how accurate the frequency vector may be, it does not reflect the best performance that would be possible with increased parallelism. If two primitives can be executed in parallel, then the execution of the slower primitive can be overlapped by the execution of the faster primitive, and thus would not be a factor in the time to complete the process step. There are two kinds of parallelism possible. The first kind arises when two (or more) different primitives use different resources and have no precedence constraints. (Obviously, any primitive which must precede another primitive cannot be executed in parallel with it.) Besides the already-present precedence graph, this requires knowledge of the resources used by each primitive. However, this parallelism comes "free": it is available on the barest hardware system possible.

The other kind of parallelism requires duplicate resources for parallel execution of primitives which have no precedence constraints. These may be different primitives or different executions of the same primitive. A fairly simple characterization of the latter case would be a companion vector to each frequency vector, containing some indication of the amount of parallelism possible for each primitive. Each value in the parallelism vector could be the number or percentage of each primitive which can be executed in parallel, where a low degree of parallelism would indicate that additional processors for those primitives could cause little improvement. However, a high degree of possible parallelism in a given primitive would not necessarily mean that additional processors would increase the speed of the function's execution, because that primitive might not be the limiting factor for the execution time.

Finding the limiting factors requires both the precedence graph of the primitives of the function, which is known, and also the relative times for the execution of each primitive, which we will call the time vector. The values of this time vector, however, are not determined until the actual implementation of the process in hardware. Yet one could make an estimate -- a hypothesized time vector. Based on this hypothesized time vector, one could determine the limiting path in the execution of each function, and the limiting path of all the functions of the process. Thus the hypothesized time vector could be used to forecast the performance of the process to see if it would meet the performance requirements. A forecasted performance much better than that required would create a high level of confidence that the performance requirements could actually be achieved, if the hypothesized time vector is reasonably close to the actual time vector finally derived. Perhaps several hypothesized time vectors could be analyzed, to produce a range of execution times for key primitives.

This analysis of the maximally parallel execution forecasts the optimum performance in an infinite-resource system. The final optimization of the application process into an optimum procedure requires knowledge of the actual time vector of the primitives' execution times, not just the hypothesized time vector. We must now go down to the hardware system and back up before this final optimization and translation is made.

4.7.3 The Design Phase of the Design Cycle

Figure 13 shows these design and optimization phases of the design cycle.

A design for which the analysis forecasts acceptable performance is feasible only in the context of the assumptions made. In order to enhance the probabilities of these assumptions being correct, the hypothesized time vector can be given as a performance requirement to the virtual operating system process. The frequency vector and parallelism information can also be given to the virtual operating system process as information which may be relevant in certain design decisions.

Going from the virtual operating system to the system which implements it involves a similar method. From the performance requirements passed down from the application process, and from the specification of the virtual operating system process itself, can be derived the performance requirements for the virtual operating system process, which will in turn be passed on to the physical operating system process. Similarly, these performance requirements for the virtual operating system along with the specification of the

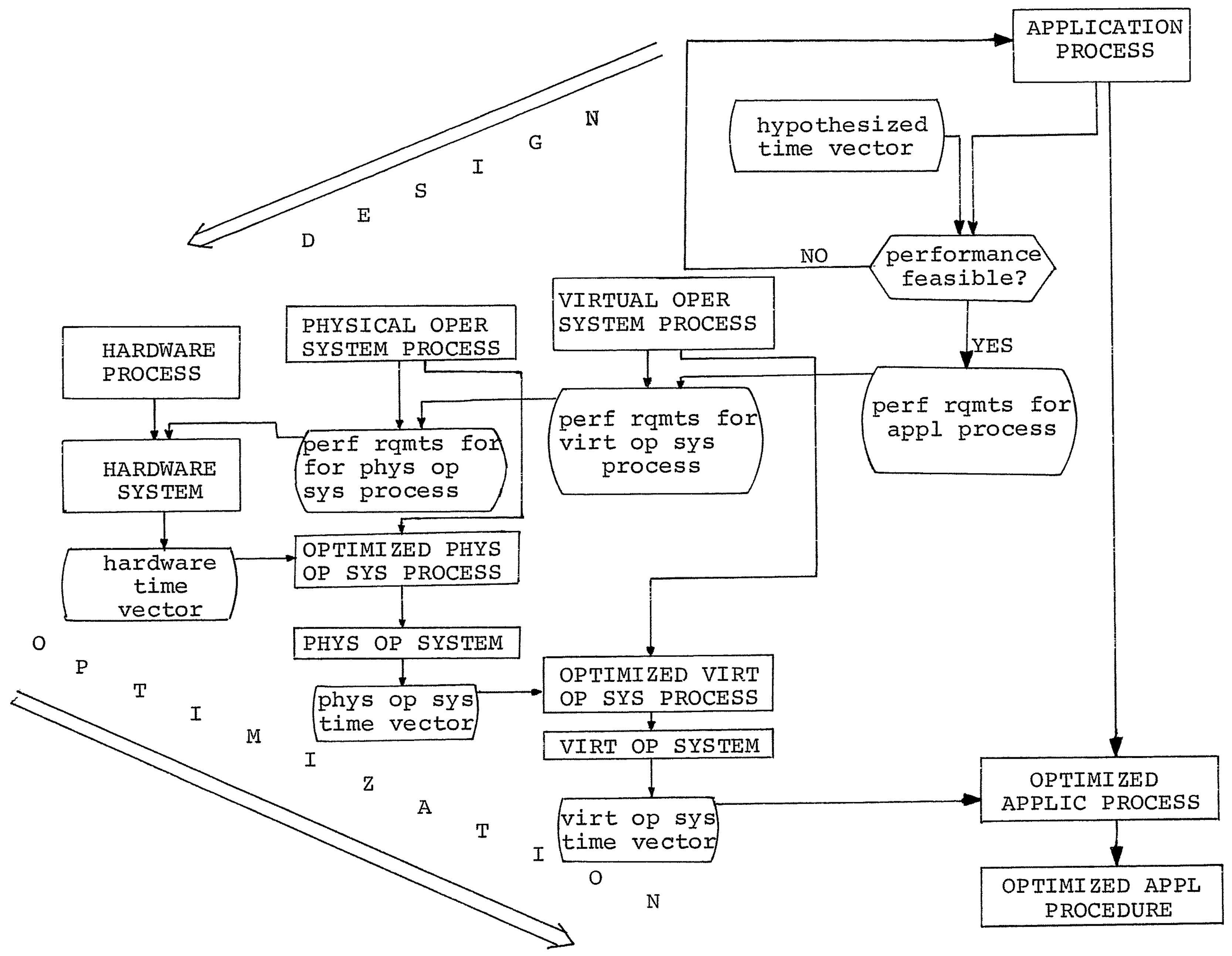


Figure 13: Design and optimization phases of the design cycle.

physical operating system process determine the performance requirements for the physical operating system process. From these and the hardware process comes the hardware system, where the final hardware time vector is determined. This ends the design phase of the design cycle.

4.7.4 The Optimization Phase of the Design Cycle

The hardware time vector now available supplies the information needed for optimization of the physical operating system process. From this optimized process the physical operating system is developed, yielding the physical operating system time vector, one level higher than the hardware time vector. Similarly, this time vector, coupled with the virtual operating system process, yields the optimized virtual operating system process, the virtual operating system, and the virtual operating system time vector. Now that the actual time vector is available, the application process can finally be optimized and then translated to a procedure. This ends the optimization phase.

4.7.5 The Integration Phase of the Design Cycle

Figure 14 shows the integration phase of the design cycle. The procedure is now integrated into the virtual operating system, providing the initial states of the system which will interpret it. The maximally parallel process, which is the optimum for the infinite-resource system, must

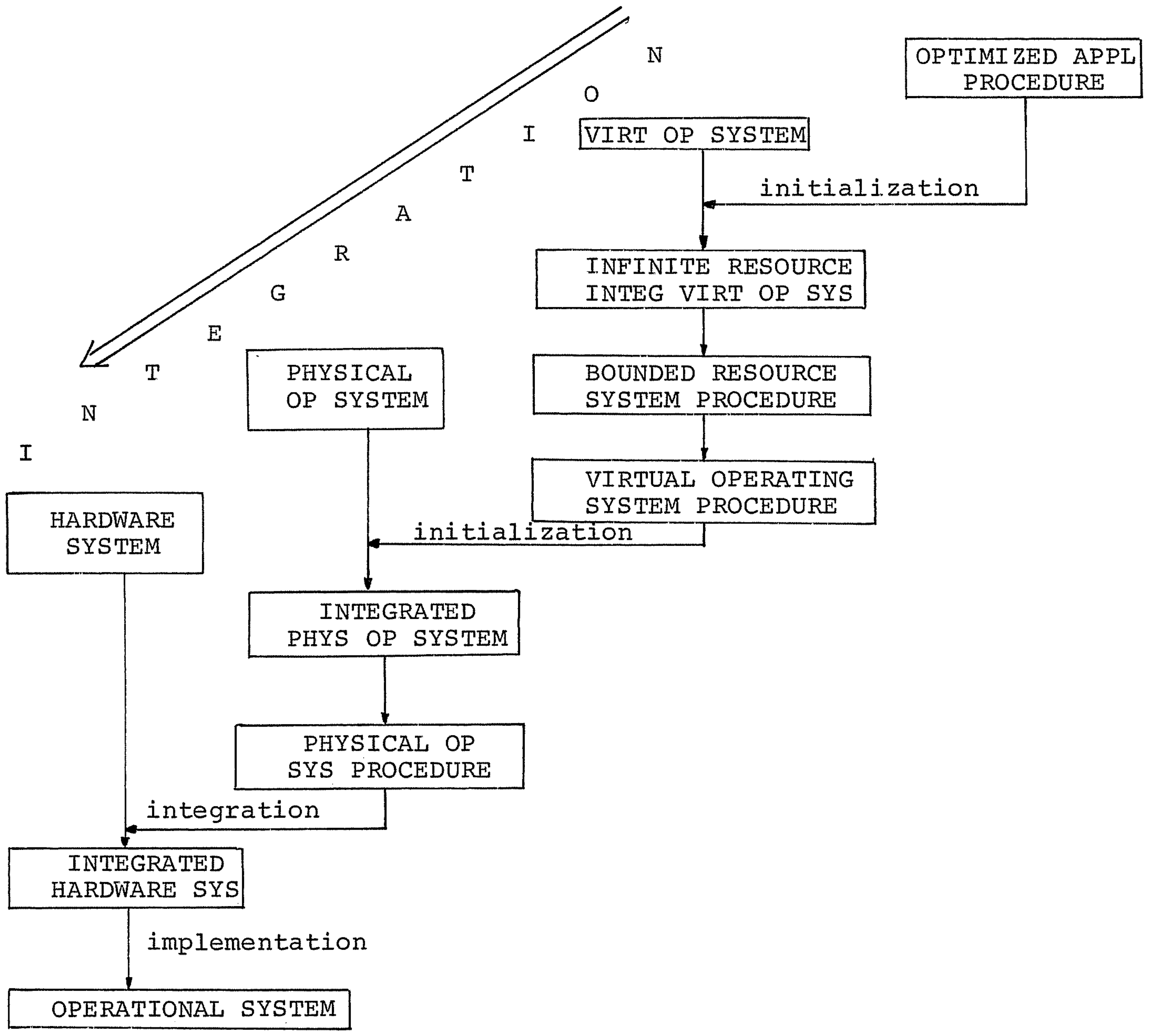


Figure 14: The integration phase of the design cycle.

now be shoe-horned into the optimum bounded-resource system. Here there are two cases: one in which the resources available are already fixed, and the problem is finding a system which meets the resource constraints and whose time is satisfactory; and one in which the problem is finding the system which meets the time constraints with minimal resources.

The first case corresponds to shoe-horning a system onto a currently available machine. Here there is no advantage in using fewer resources than are available, but it is impossible to use more. (This ignores any differences in the difficulty and thus time of scheduling with increased loading. This also ignores any constraints on the use of the resources to allow for future growth, but these constraints could be defined by specifying fewer resources than were actually available. For example, if only 50% of 100 K of core may be used, one could simply specify that only, say, 52 K of core is available.) In this situation it would be advantageous to double the amount of a resource used (if that extra were available) in order to save a small amount of time or to free a different resource for which the demand exceeded the supply. The time for a process step would be computed similarly to the infinite-resource system, except that serialization would be forced not only by precedence constraints but also by resource constraints. Any place at which a resource is underutilized would be a potential place for optimization.

The second case reflects the situation in which the hardware has not been specified. Here trade-offs among the different kinds of resources must be considered. The cost of adding a given resource would be balanced by a greater savings on another resource, or could be justified by additional parallelism resulting in a savings of time. This case is more complex than the others, because the cost of the resources as well as the time is being minimized.

In the limit this second case of the bounded-resource system reduces to the infinite-resource system, where enough resources have been added to be able to exploit all possible parallelism.

Do we really need to find the optimum bounded-resource system? The answer is no: we only need a system which is good enough to meet the time constraints imposed by the designer (and the cost constraints, in the second case). Any system which can be shoe-horned into the available machine and provide the required response time is adequate; there is no need for additional optimization to further reduce the time.

Similarly, when building a machine by adding resources, once the required time bound has been met, there is no need to increase the cost by adding resources to increase the speed.

From the bounded resource virtual operating system one can develop the procedures which the physical operating system will interpret. This procedure then is used to initialize the physical operating system. Similarly, the physical operating system is translated into a procedure which is integrated into the hardware system. The hardware system is finally implemented in the operational system. This is the end of the integration phase and of the entire design cycle. At this point the application process is actually represented by a program running on a physical processor with given resources.

4.8 Conclusions

Our functional specification of interacting process clusters can be used to formulate real-time system problems and to test proposed design laws which would make specified systems testable. A significant part of the logical testing for correctness and boundedness can be carried out prior to real-time testing in order to simplify the required real-time testing. Some desirable properties of resource management have been identified. Only the briefest sketch of the dynamic models that must be developed has been given. A substantial amount of work remains to be done to complete this study and to demonstrate the practicality and usefulness of the design laws. Some further work on mappings between application processes, virtual operating systems, and physical operating systems is discussed in the next section.

5. DISTRIBUTED DATA PROCESSING SYSTEMS

The requirements specification process was discussed in Section 2. Here we will further develop that section's ideas concerning system (and requirements) decomposition/-integration, in the context of the design of a distributed data processing system, and present an example of a distributed system design. It should be noted that the system example presented in this section was designed before many of the concepts discussed in Section 2 were formalized and in fact served as a motivating force for some of that work. Thus the purpose of the inclusion of this example in this document is not to claim that this is the "best" distributed data processing system but rather to illustrate the effect of some of the ideas of Section 2 on a particular design process.

5.1 Introduction

Distributed data processing systems encounter problems which are unique among computing systems. Not only are the problems common to large scale computing systems present, such as the effective use of system facilities by the users of the system and software redevelopment in response to changing technology, but also the problems created by an often very large class of users with widely differing application needs. Problems also arise from the desire of users of a network to communicate with each other to an extent not found in conventional systems, in order to gain information or share resources. With the added probability of errant or malicious processes, it seems that the conventional centralized operating system is inadequate. Thus the network designer is forced to look toward a decentralization of the network operating system in order to solve his problem.

This section will develop ideas about functional requirements specifications and their design process which may be used in solving these problems of distributed data processing system design. Design decisions involving decomposition and integration must be based not only on functional requirements, but must also reflect resource and performance requirements optimization. This section will also discuss an example of such a decentralized network system which was originally presented in [Kra73]. The discussion will deal with both the design process and the design itself. Section 5.2 will develop system decomposition/integration ideas. Section 5.3 will discuss the design process of the network example to illustrate some of the system decomposition/integration ideas. Also in Section 5.3 will be discussed the design constraints postulated prior to the design and their effects on system decomposition/-integration. Section 5.4 gives a brief description of the

---

[Kra73] Kramer, John F., _A General Structure for Uncooperative Processes Distributed over a System Network_. Ph.D. Thesis, University of Wisconsin, Madison, 1973.

network itself and discusses some design decisions as they relate to Section 5.3. A more complete presentation of the design of the network is presented in Appendix C. Section 5.5 will deal with the generality of the resulting network, and Section 5.6 will summarize the discussion of Section 5.

5.2 System Decomposition/Integration

Clearly, in order for the design process to be manageable, it must be decomposed in such a way that most decisions can be made locally, based on data available within a local area of the developing system specification. As discussed in Section 2, decomposing the design process may be done by identifying the "tightly coupled" attributes of the system and factoring the system with respect to them. In order to avoid the requirements allocation problem, a system must be factored into subsystems in such a way as to allow the functional (logical), resource, and performance requirements to also be factored over the subsystems. It is also necessary that design decisions affecting requirements allocation be made at those points in the design process when there is sufficient data on which to base them, and that the effects of these decisions be displayed to the designer.

The design process develops a functional specification of the application which is consistent with and motivated by the functional requirements. There is also a specification of hardware which is consistent with the resource requirements, and a mapping of the functional specification onto the hardware specification which, combined with the hardware performance parameters, must be consistent with the performance requirements. The mapping from functional specification to hardware may itself need a complex functional specification, which is factored in the system in the form of a virtual operating system specification. We say virtual because many of its components are abstractions of the physical resources of the hardware specification; these abstractions are virtual resources. Thus it is natural to factor the system into application system, virtual operating system, and hardware system specifications, along with mappings from application to operating systems, and from operating to hardware systems. This achieves a natural factoring of the concerns of the designers:

1) In the application system specification the state domains and the state transition functions are problem oriented, and are guided by the functional requirements.
2) In the operating system specification the state domains reflect abstract resource types and the state transition functions reflect resource use and access methods, and are guided by functional and resource requirements.
3) In the hardware system specification the state domains reflect physical resources and the state

transition functions reflect the logic and interconnection of physical resources, and are guided by resource and performance requirements.

4) The mappings from application to operating to hardware systems specifications reflect resource (virtual and physical) use and sharing, allow behavior and performance effects of sharing to be displayed to application and operating systems designers, and are guided by interactions of requirements.

Thus we have a start on avoiding the requirements allocation problem.

We have defined a formal notation for system specification in Section 3 but we have no formalism developed for resource mappings between system specifications. Thus we should get some idea of what they look like.

A system specification is a set of interacting processes containing state spaces and state successor functions which are elaborated into state components, more primitive state transition functions, and intermediate state domain and range sets for the primitive functions. In terms of resources these logical structures need storage types for state spaces and function domains and ranges, and they need processor types for primitive function evaluation. Primitive exchange functions need channel-processor-type resources. There must also be some type of control in the resource specification into which we may map the precedence constraints of the primitive functions, the synchronization of exchange primitive functions, and additional precedence constraints needed to avoid resource conflicts generated by shared resource mapping. These resources and control may themselves be encoded by our notation for functional specifications, and the resource mapping is then from one system specification to another. These mappings may be accomplished with a wide variety of structures. The mappings may be dynamic, with changes bound either to actions (function evaluations) in the domain (logical system specification) or in the range (resource system specification) of the resource mapping. There is also a spectrum of how dynamic the mappings to each type of resource may be; references to a logical structure may always be mapped to the same resource structure or they may require a computation of which resource structure they map to. When the resources are core and an instruction execution processor, this spectrum corresponds roughly to that from compilation to interpretation. Also different mapping structures may be employed for different resource types, for different logical components mapping to the same resource type, and for different granularity of a resource type. These differences in resource mapping structures occur because decisions about them are made at different points in the design process. Thus a resource mapping is a varied structure which reflects the overall design process. These remarks apply to both the mapping from application to operating system specification and from operating to hardware system specification.

The decomposition of the system specification may be continued by identifying "tightly coupled" attributes within the functional requirements and splitting the functional specifications into local areas which are developed separately. If this splitting is done in a "top down" manner during the design process, we may develop a natural correspondence between local areas of application, operating, and hardware systems based on the mappings between these systems. This is the decentralized processing idea; a group of similar or cooperating application processes are served by an operating subsystem which in turn is implemented on a hardware subsystem, and the designs of application, operating, and hardware subsystems have been coordinated. In order for this type of decomposition to yield more efficient design steps and ease the problems of subsequent integration steps, it must be built into the design methodology. This can be illustrated with the example in Figure 15. Starting at the original system specification the design process may go as follows:

Step 1) Elaborate application system into three subsystems, with decisions based on the functional requirements.

Step 2) Elaborate operating system into three subsystems and elaborate mapping from application to operating systems, with decisions based on types of

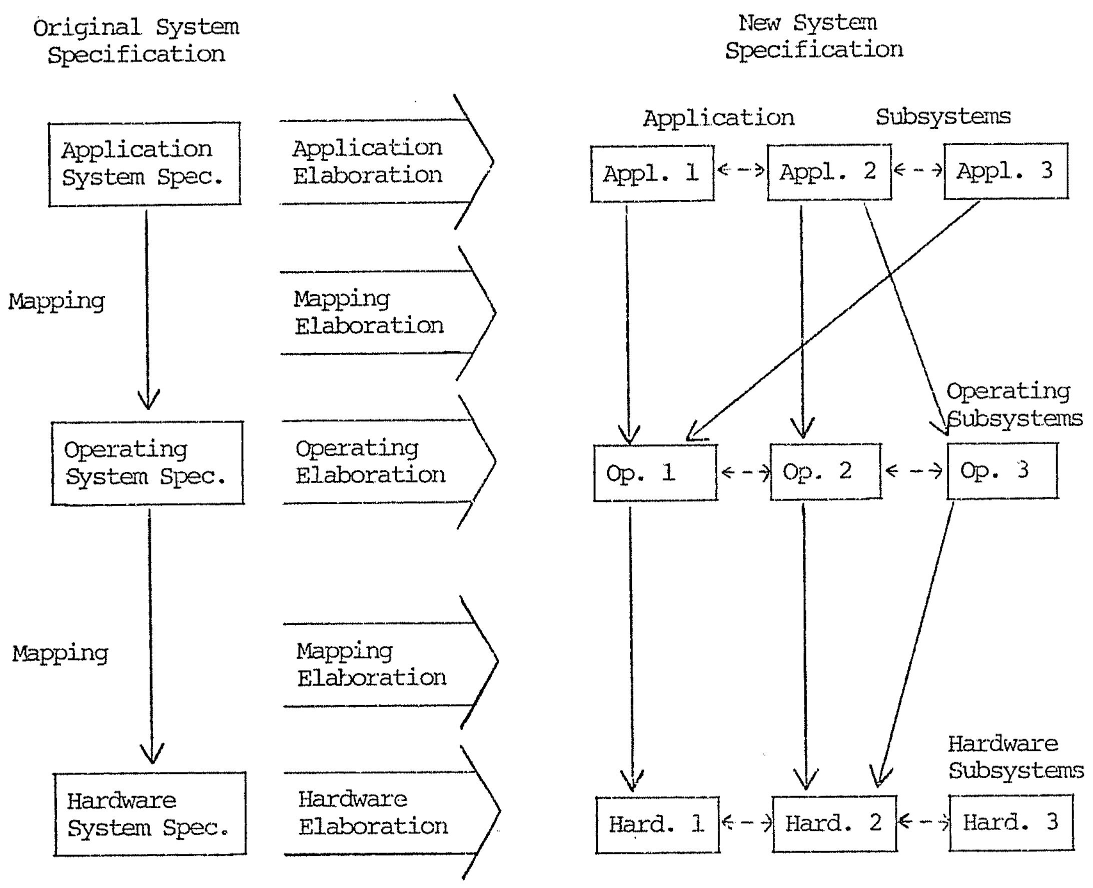


Figure 15. Dotted arrows represent subsystem interactions, solid arrows represent mappings of a specification at one level onto a specification below. This example illustrates the design process decomposing the system specification in a "top down" fashion.

resources needed by application subsystems and on types available in resource requirements.

Step 3) Elaborate hardware system into three subsystems and elaborate mapping from operating to hardware systems, with decisions based on resource and performance requirements.

Step 4) The elaborated operating system and mappings introduce new resource-sharing effects in the form of added precedence constraints in application system behavior and in the form of performance changes. These effects must be calculated and displayed to the appropriate designers, allowing them to return to steps 1), 2), or 3) to alter design decisions; it may be necessary to make an alteration in order to meet functional and performance requirements, or it may be possible to make further optimizations of resource sharing.

The decomposition of design achieved here serves to separate further design steps for the new subsystems, and most interactions of these design steps will occur between a subsystem and the subsystem(s) below, into which it is mapped.

The type of design step indicated in step 4) above is important for system integration, which can account for a very large percentage of total system design effort [Ste76].

[Ste76] Stephenson, W. E., "An Analysis of the Resources Used in the SAFEGUARD System Software Development," Proc. 2nd International Conf. on Software Eng. San Francisco, 1976.

The effects of resource mapping decisions must be displayable to and controllable by the appropriate designers. These effects include added precedence constraints affecting behavior and changes to performance; if these effects are not controlled at the appropriate point in the design process they will make system integration very difficult, perhaps necessitating returning to the appropriate point in the design process and redoing the design. The display and control of these effects can be computationally infeasible if arbitrary resource mappings are allowed, so we need to find constraints on the mappings which ensure that display and control are possible.

Displaying the effects of resource mapping will be easier if the mapping is predictable. For example it is difficult during design to see the effect of a paging scheme on an application system because it is hard to predict whether at a given time an address will be in core or on disc. However with an application-controlled overlay scheme we can say much more about the behavior the system will have because we can predict where logical addresses will be at points in the computation. Motivated by this example we may find a number of canonical structures for mappings from application to operating systems which make it possible to predict, at a given point in a functional (logical) computation, which resource a functional domain is mapped onto. In these structures, changes in the resource mappings would be bound to points in the logical computation, and scheduling necessary to avoid resource conflicts would be bound directly to the logical computation. We refer to these structures as functionally bound mappings. This idea also applies to mappings from operating to hardware systems. With these functionally bound mappings, processes in the lower level specification are slaved to processes in the higher level. However there may be processes in the lower level which are not controlled from above; these may complicate the task of displaying effects of resource sharing. For example in an application-controlled overlay scheme, logical addresses may be mapped into virtual core but there may be an independent operating system process which controls mapping of virtual core into a combination of physical core and physical disc. Thus in order to display and control effects of resource mapping, we need canonical structures for operating and hardware systems specifications. These structures do not necessarily need to be as restrictive as functionally bound mappings, but they must allow some control over resource mapping at the functional (logical) level, even if this control is implicit. For example we may allow a paging scheme mapping if the application designer has adequate tools to control his "working set" and "hit rate". A primary criterion for the canonical structures is their ability to ease system integration by display and control of the effects of resource sharing, and functionally bound mappings are the baseline for this criterion.

Estimates of system performance are made by starting with hardware performance parameters, going back through the mapping to operating system functions and estimating their performance, then going back through the mapping to application system functions and estimating their performance. This way it may be difficult to get a good estimate, depending on the resource mapping structures and on the complexity of application and operating system functions. However we may be able to approach performance estimation by a combination of simulation testing and reasonable resource mapping structures. These reasonable structures must have a monotone decreasing performance when demands for resources increase; in fact we might ask for performance to be a convex function of resource demand, with performance zero at infinite resource demand. Then, given a system with resource sharing, we could do a number of simulations in order to display to the designer some bounds on performance as a function of system load parameters. These ideas are discussed more in section 4. They suggest useful criteria for canonical resource mapping structures.

We have seen in connection with Figure 15 that we can decompose the design process and make decisions in local contexts. However there are nonlocal optimization and integration steps which must be accommodated by the design process. For example, several local areas may discover in elaborating their resources that there are points in their computations when not all their resources are needed. For them to share these resources requires nonlocal optimization, with nonlocal effects. Decisions about these effects must be made by an authority not present in the local design areas. Thus when the design is decomposed we must specify an authority which can execute design steps making nonlocal decisions to avoid integration problems. Furthermore the decisions of these design steps will be reflected in canonical resource mapping structures which interact with the structures of local specification areas.

Further development of these ideas must be done in a more formal context. We need to:

1) Find a formal notation for specifying resource mappings between system specifications.
2) Find algorithms and methods for displaying precedence and performance effects of specified resource mappings.
3) Find canonical specifications for resource mappings for which the algorithms and methods above can provide useful information.
4) Find canonical specifications for operating and hardware system processes for which we can usefully display their precedence and performance effects on the application system.
5) Discover which of these canonical structures are reasonable in the sense that performance is a monotone decreasing function of resource demand.
6) Incorporate these formal tools into design steps for inclusion in the developing formal design methodology.

5.3 Design Constraints of Network Example

A key motivation of the network example is avoiding the requirements allocation problem by factoring the system in such a way that the functional (logical), resource, and performance requirements may be factored over the subsystems. To ensure such a requirement factoring, Kramer decomposes the operating system into two operating systems, the virtual operating system and the physical operating system. He now can present to the applications designer an abstraction of the system in which the designer sees only the virtual operating system and can specify his processes in terms of virtual resources and virtual operating system primitives, without any concern for the hardware implementation of the system. Similarly, the hardware designer's view of the system is application-independent: his only concern is to implement the physical operating system. The physical operating system designer's problem is now also clearly defined. He must map the virtual resources onto the physical resources. Since all three designers have nearly independent views of the system, minimal constraints are placed on them in their desire to optimize their subsystems to their own performance, resource or logical requirements.

The method of specification which the applications designer uses to specify his processes can be that which was described in Section 3. The same is true for specifying the virtual operating system operations. An example of this type of specification for the Kramer network system functions is given in Appendix D.

The system integration phase can occur when the applications designer has specified his processes in terms of virtual operating system procedures which can then be compiled or translated into physical operating system procedures. The physical operating system procedures can then be compiled or translated into machine code which can be executed. This process is illustrated in Figure 4 of Section 2. It is not clear at this point whether system integration must take place in this manner or whether something on the order of a purely functional specification of the application process can be "compiled" into some specification of the physical operating system.

Even though a comparatively large amount of freedom has been given to the individual designers to optimize their own subsystem, there is no guarantee that all the requirements will be met on system integration, which creates the possibility of a design feedback loop.

Once Kramer has established the factorization of the system, he postulates the following design constraints:

A. Network Communication

"The designed network must consist of a set of interacting, bounded, programmable, digital computer systems with well-defined processes, and be open to communication with foreign systems having undefined processes."

B. Responsibility factorization

"Responsibility for meeting the postulated design constraints and any design decisions must be factored between the network, implementation, and process designers."

C. Authority

Each designer must have sufficient authority to define the effects, on the processes running in the network, of the interactions for which that designer has been delegated responsibility."

D. Delegation

"Although processes must be permitted to delegate to other processes their authority to control process interactions, the delegating process always retains the responsibility for that interaction and the authority to control the process to which it was delegated."

E. Modification

"The network definition must provide for modification of its definition. The design must provide sufficient constraints to be able to delegate safely to process managers all modification decisions and authority to control the effects of such decisions on the processes."

The design constraints fall into two categories:

(1) Those constraints which ensure some desired network characteristics: constraints A, D, and E.

Constraint A requires that the network integrity must be maintained in terms of its own processes and its interactions with foreign processes. Constraint D requires that no design decision is made which arbitrarily centralizes authority, thus ensuring a maximally decentralized network. Constraint E provides a means for network evolution to meet future needs.

(2) Those constraints whose only purpose is to preserve the decomposition obtained at the beginning of the design process: constraints B and C. Constraint B states that each subsystem designer will have the responsibility to meet the other design constraints as they apply to his subsystem, and constraint C requires that each subsystem designer will in fact have the authority to do this. As an example, it would be a violation of the decomposition if the applications designer could specify as part of his application exactly how the virtual memory he used was to be mapped onto physical memory, as this is under the realm of the operating systems designer. Consequently this action also violates constraint C.

5.4 Brief Overview of the Network Design

This section will present a condensed version of the design of the network presented in Appendix D, after which some of the design decisions will be discussed in terms of the design constraints presented earlier.

5.4.1 System Structures

If the design constraints stated in Section 5.2 are to be useful, some formal definitions are necessary. A digital system is composed of two parts, a _system processor_ and a _system state_. Operators are applied by the system processor to define a new system state or to construct messages for transmission to other system states in a set of digital systems. The execution of a system processor occurs in two distinct phases. Given an initial system state, the processor evaluates all operators which apply and produces a new local system state. Each operator can be executed in parallel and without side effects on the other operators being executed. The local system state is then updated by any messages from other system processors, and the local interpretation cycle begins again. This sequence is called a _system step_ and transforms the system state into its successor state. This discrete sequence of states is called a _digital computation_. A set of digital computations represents a _digital process_. The interpretation cycle of each system in a set of interacting digital systems is independent and asynchronous. All inter-system communication will be defined in terms of asynchronous message transmission with any desired synchronization explicitly carried out by the cooperating digital processes. Messages will be received, after a finite delay, in the same order as they were transmitted by a given system. Some particular digital systems are defined as network systems, and a set of such systems as a network. There is a universal simulator of such sets of digital systems. Thus the system specification is already in a form that can be studied and tested. There are no physical implementation constraints on how many network systems may be implemented on a single physical system or on how many physical systems are used for a network or network system.

The network system states are defined as a set of elements. Each element can be interpreted as the state of a synchronously interacting process component of the overall system process. All system state information and all system computations are based on these system process elements. Network system operators are constrained so that at most one operator will modify a given system state element in a given interpretation step.

Although we wish our system processors to be well-defined with respect to their operations on the system state, we desire also to permit them to communicate with systems outside of the formally defined networks which are not well-defined. Such systems are called foreign systems and are not constrained in their internal structure by the network designers, except for a communication conventions they must use if they wish to interact with a network system. Certain elements of the system state are housekeeping processes that manage inter- and intra-system interactions. These elements contain system state information and are neither explicitly transformed by application programs not transportable between systems in the network. Those elements of the state that are explicitly programmed, transformed, and moved from system to system will be called network (i.e., user) processes. A network process step consists of a cycle of three system process steps or phases. Two of the system phases are used for housekeeping functions and will be described later. Operators applied during phase two of a network process step will be called system subprocessors; their effect is local to that network process state. The complexity of a given network process step is therefore defined by the corresponding subprocessor. The process state defines the current address space of the process. Figure 1 in Appendix C is an informal characterization of one network system.

In review, a foreign system is not characterized (or constrained) except as a source or destination of messages. A network system is composed of a system processor and a system state. A system processor is composed of a set of operators, some of which are subprocessors. The system state is composed of system housekeeping elements and system elements interpreted as network (user) process states. A network consists of an intercommunicating set of such systems and foreign systems.

5.4.2 Network System Interactions

The mechanism for inter-system communication selected here is quite similar to the notion of a hierarchical interrupt system. In phase one, the system processor determines which subprocessors (if any) should be applied to each network process state in that system and delivers the appropriate messages (if any) to an interface buffer in each network process state. In phase two, the system processor applies the selcted subprocessors to the selected network process states thus causing them to undergo a network process step. In phase three, the system processor performs the nonlocal services requested (if any) by messages left in the interface buffer by a subprocessor. Nonlocal operations are those which have side effects external to a network process state. Messages may be received by the system processor in all phases and are buffered until their delivery in some subsequent system phase one.

The system processor must be able to recognize, in any network process state, which subprocessor to apply in phase two, and each subprocessor must be able to recognize its own state information component in that network process state. For these purposes, each network process state will contain one or more objects called control points. Although control points are used for a variety of purposes, only those aspects affecting subprocessor application will be described here. A control point roughly corresponds to an interrupt level in conventional systems.

All control points in a network process state are ordered by priority within that state. Each control point has an ordered set of channel terminations and buffers associated with it in the system state elements for the receipt of messages. Depending upon the status of the control point buffers and also the status of the control point within a process state itself, messages may or may not be received by the buffers, and the control point may or may not be eligible for subprocessor allocation during a subsequent phase two. The details of message delivery are covered in Appendix C. The highest priority control point candidate present in a network process state will be made active, and the pending message, if any, will be placed in the process interface buffer.

During phase two, the appropriate subprocessors are applied to network process states containing the active control points selected during phase one. At most one subprocessor is applied to a given process state during any phase two step. Although many process states in a given system may be requesting the same subprocessor, they each have it applied in parallel during phase two. Whether they are really serviced in sequence or in parallel is an implementation decision that affects only the duration of phase two since no interprocess interactions in this network system can occur until phase two has completed. All processes containing active control points will have subprocessors applied prior to the end of phase two.

Each subprocessor transforms only the network process state to which it is being applied. The applied subprocessor thus defines the successor network process state. If a system service is requested (a nonlocal transformation), the subprocessor will complete its network process step leaving a message requesting the service in the local interface buffer for subsequent phase three processing. Note that network processes in the same system will proceed in <u>synchronous parallel</u> with each other, each completing one process step in each system cycle. Network processes in different systems will run <u>asynchronously parallel</u>.

During phase three all requests for nonlocal services which were left in the interface buffer are acted upon. There are four types of these services: message transmission, resource transmission, process state transmission, and system modification. Phase three ends when all such services have been completed and the interface buffers are cleared.

If the interface buffer contains a request for message transmission, the associated message is removed from the buffer and "broadcast" to all systems but addressed to a particular control point buffer. The system containing the control point (destination) buffer will receive it; other systems will ignore the broadcast. See Figure 2 in Appendix C. The message has a network standard format and is represented as a string over a network standard character set.

Although messages broadcast during phase three are subject to unspecified transmission delays (unless messages are intra-system), the order of transmission between any two systems is preserved in perception. In the receiving system, messages are placed in channel termination buffers and used to update control point buffers during every phase. As far as network processes are concerned, message transmission is transparent to system boundaries, and network processes may cooperatively move from system to system without losing communication or restricting their interactions.

A network process defines the local environment and address space for subprocessor transfromations. The interface buffer serves to factor the subprocessor transformations which are local to the process state from the system processor services which have effects outside of the process state. A process state will contain one or more control points, the status of which may be modified during system phase one as a result of control points in interprocess communication. Control points also serve to delimit uniquely a portion of the network process state, called an <u>expression state</u>. Within this portion of the process state the designers of the corresponding subprocessor are responsible for defining both the representations and the transformations which that subprocessor will carry out on those representations. All of the state information required for a subprocessor to continue a computation must be part of any corresponding expression state. A network process state thus contains an interface buffer, a set of expression states and one or more control points.

Many freedoms are given to the network process designer involving complex process interactions, including manners in which to deal with uncooperative processes and to exploit known instances of cooperation. These are detailed more fully in Appendix C. In short, a single network process is capable of supporting all computations that can be carried out on a conventional single-processor, multiprogramming system.

There is a subprocessor, called GP (General Programmable), in every network system. The GP expression state can be programmed, in the network system language, to provide on request all network services. GP expression states thus can specify invariant computations despite movement of the containing network process. The network system language thus plays many roles such as the following:

a) The network job control language.

b) The network high level "machine" language.

c) The network implementation language for operating "systems".

d) The base language for definitional extension via source language macros or compilers.

e) The base language for evolutionary augmentation.

f) The system invariant computation language.

The problem of deciding when a given interaction is improper can only be resolved by another process aware of the interactions. The system can not resolve conflicting claims of one process with respect to the behavior of another process since such resolution may require intimate knowledge of the specific applications. In order to guarantee resolution there must be a responsible authority who can control and manage interactions between the warring parties. Our system imposes a hierarchy of uniquely designated responsible authorities in the form of a network process tree, as in Figure 4 in Appendix C, to make such an arbitration. All authority rests initially in the root of the tree. Although delegation to a child process is allowed, each root of a subtree remains responsible to its parent for the interactions of the processes in that sub-tree. An uncooperative root of a subtree will still be accountable to its parent process. To insure this accountability the network provides another subprocessor, the AO (Accountable Object) subprocessor. The details of AO are available in Appendix C.

The process tree can both grow, by creating new "leaves", and shrink, by destroying "leaves". An existing process may move, as permitted by the parental authority, to any network system known to that parent. Thus both the process tree and its distribution over the network systems can change dynamically as a result of process computations. Since these are intrinsically non-local operations, the GP sub-processor can only request them. The responsibility for carrying them out has been delegated to a PR (Process

Receive) process and its corresponding PP subprocessor. Each non-foreign network system will contain one of each. The GP requests for such service thus take the form of messages to the local (to the network system) PP process. Process birth and death and further details of the PP sub-processor are available in Appendix C.

5.4.3 Design Decisions

As was indicated in Section 5.2, allowing applications designers the ability to specify their processes in a hardware independent manner accrues many advantages. Hardware changes are allowed to take place without the necessity for rewriting applications, allowing a consistant virtual "machine" throughout the network which permits a process to move from one system to another to exploit special hardware features. Perhaps more important from a design standpoint, the network remains clearly factored into the subsystems described in Section 5.3, allowing for local optimization to take place at all levels.

5.5 Generality

The system postulated in 5.3 and described in 5.4 seems general enough for a large variety of applications and, with the ability for self-modification, it would seem it could meet future needs. In addition to the generality it does allow for specialization in terms of special purpose hardware systems.

The design process and the resulting network seem to indicate that many of the problems inherent in a distributed data processing system can be dealt with effectively in a decentralized network operating system environment.

5.6 Summary

The design process of the Kramer system could be summarized as follows:

1) Decompose the system into subsystems over which the requirements can be factored. This necessitates the division of the operating system into a virtual operating system and a physical operating system.

2) Postulate design constraints so that subsequent design decisions in all sub-systems will

   a) preserve the external requirements, i.e., well-definedness, provision for evolution, and decentralization.

   b) preserve the decomposition obtained in 1.

3) Develop the design or specification of the subsystems to the point where integration can take place, i.e., the translation of application processes into virtual operating system processes and again into hardware processes, as illustrated in Figure 4 of Section 2.

Once again, to the extent that (1) is accomplished without side effects, (3) will be correspondingly easier. This design process allows for local optimization of subsystems as well as local testability, in the sense that application processes can be tested on a "virtual" machine independent of implementation, and the mapping of virtual resources onto physical resources can be tested in the absence of particular application constraints.

Our functional specifications can describe the resulting "Kramer Systems" and processes. The need for decomposition into application (again decomposed if needed), virtual operating, and physical operating systems lead to additional "good design" principles for requirements specification, i.e., state requirements in terms of such models.

6. THE DESIGN SCHEMA

6.1 Introduction

In this section we deal with the methodology of system design primarily in the context of the formalism developed in the previous sections. We are interested in a general strategy, more formally a design schema, to be realized as an advanced set of tools for the systems designer. Our development of this strategy is intended to be as unbiased as possible with respect to the particular aims of a given designer or to the ultimate application of the systems which he specifies. Further, we want to restrict ourselves to design techniques which maximize the possibility of local analysis and testing of system modules throughout the course of a top-down system definition. Thus our purpose here will be to explore those properties which we are already prepared to require of the proposed design package in order to identify the directions of future research in this area.

6.2 Design Processes

A key concept in this discussion is that of the design process, which is the second state in the development process. In the most general sense we can define a design process as a sequence of analyses, decisions, and commitments (all design steps) which start with a formal requirements specification and lead to a formal system specification implementation. As discussed in section 2, there is substantial overlap between the requirements process (which may also carry out part of a design process) and the design process itself. Much of the discussion of the formal requirements process steps in section 2 is directly applicable to this section. Obviously there are many design processes which can lead to a single product, and many products which can satisfy a given set of system requirements. Moreover we may characterize these design processes grossly in terms of such factors as cost, total human effort expended, and success of the product in meeting requirements. However, it becomes apparent that we must find it difficult indeed to model a design process after the fact if the sequence of analyses and decisions mentioned above is poorly structured, for example, as with decisions made by the designer with an uncertain or imprecisely stated impact on future decisions. We could (and in current practice usally do) have design decisions which are recorded only as informal statements or which are presented in no tangible artifact at all. Furthermore, given poorly structured design techniques, if we choose to restrict the evolution of design processes in order to control some function of the contingencies of the design steps (such as cost or time incurred in executing the product software), then we face an even more complex problem.

Since our goal is in fact to provide design laws which will allow us to satisfy system requirements, we are inevitably led by the arguments of the preceding paragraph to demand that steps in the design process step can be determined by criteria imposed by the design laws and systems requirements. Viewed in this respect the design process shares features with the computational processes described in earlier sections. In particular, we have functional specifications as the states of the design process, transitions between these well-defined states (carried out by the designer-aided digital computations), and finally possible interactions between asynchronous design processes.

The major difference between design processes and the processes specified in section 3, apart from the difference in state values, is that in the former processes, state transitions (design steps) cannot be guaranteed to be algorithmic (finite) or bounded in computation time. This inability to bound computation time results from the potentially complex nature of many design steps, for which no effective procedures exist. Further, human designers can carry out design steps that no machine can do and are not absolutely required to succeed. A simple illustration would be an optimization step in which a design procedure may search in vain for an impossible optimization specified by functional requirements. Thus the need for human intervention in design steps makes a definition of these step procedures by strict mathematical algorithms unattainable. We summarize by saying that well-defined process steps, the termination of which can be decided, are a minimally acceptable framework for the conceptual and practical formalization of intended design laws.

6.3 Design Steps

Some new features which we wish to impose on design processes can best be introduced by first giving a review of the types of design steps which might occur within these processes. This cursory review is not meant to imply that to carry out a design step is a trivial matter. On the contrary, a great deal of design and computational effort may be involved. We have discussed already in section 2 the decomposition and integration of functional specifications and corresponding design processes; these will be elaborated in section 6.4. In the preceding paragraph we also mentioned briefly optimization steps, to be treated more fully in section 6.4. Interactions with other design processes, at the intermediate steps of a design process, must also be included to compensate for prior decompositions into loosely coupled requirements. The binding step corresponds roughly to implementation of some entity in terms of another, as in the binding of control to a particular sequence or in replacing a function by a procedure. In summary we have the following kinds of steps:

. Decomposition: factoring specifications

. Integration: composing specifications

. Optimization: finding a better specification

. Interaction: inter-design-process communication

. Binding: encoding a design decision by elaboration of detail.

We will require that our formal representation of processes and systems requirements be amenable to several types of analysis in conjunction with each of the kinds of steps above. In particular we want to know whether the design step is able to meet the requirements under the constraints imposed by previous design decisions. We also want to be able to subject the system to simulation or other investigation in order to proceed intelligently with further decisions. However, in the event that a design step fails, we must be prepared to perform it again with a new set of parameters, or if that fails, to retreat to some earlier design step, repeat it with new parameters and move forward again. Finally, it is apparent that if design step computations, analyses, and simulations are mechanized then much of the cost and time required in systems design can be reduced. Also we can minimize the introduction of errors and maximize the ability to detect errors.

6.4 Decomposition, Optimization, Integration

In this section we will provide a sequence of design steps which might typically be applied to a simple functional specification in order to reduce it eventually to a set of procedures for evaluating the function originally defined. First, by way of example we will deal with a single expression which consists entirely of primitive functions other than exchange functions, thus remaining at a single level of abstraction and ignoring for the moment the more complex issue of asynchronous interactions. The latter subject will then be approached within the scope of a set of design tools dealing with exchange functions. The three types of design steps which we now treat involve decomposition, optimization, and integration. Decomposition of functional specifications into modules (where we leave modules undefined for present purposes) is based here upon the ability to perform different component computations of expression evaluation in parallel, a key factor in subsequent optimization operations. Figure 16 shows the computational precedence graph for a particular expression consisting entirely of primitive functions. This graph indicates that evaluation of a parent node awaits the evaluation of its decendent nodes, and that the descendent nodes of a given parent may be evaluated in parallel. An unsophisticated approach toward creating system modules

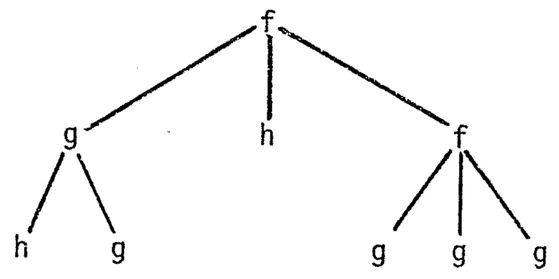


Figure 16. Precedence graph for the expression
f(g(h(a,b),g(c,d)),h(c,d),f(g(a,b),g(c,d),g(a,b)))

(that is, toward creating a new set of functional specifications which reflect the potential for computational parallelism) is to assign a module to each node in the precedence graph. In the formalism of preceding sections, this would correspond to modelling the state successor function applicable to the expression of interest by an interacting complex of degenerate system state successor functions. (A more sophisticated decomposition approach might involve a recursive procedure which could be applied to the expression in several states. The development of such an approach is a topic for further research.) We have thus described the decomposition process which comprises the first design step of Figure 17.

Our first type of optimization design step exploits computational parallelism in a simple way. It constitutes the second design step of Figure 17. For the sake of illustration let us assume that we are concerned with the evaluation of the expression f(g(a,b),g(c,d),g(a,b)), a subexpression of the one in Figure 16. Since the first and third arguments of the function f are identical, we only need to evaluate one argument and save the resulting value so that the same computation does not have to be performed for the other argument. Whether this optimization, or any other type to be discussed below, will in fact be performed depends on the functional requirements, since the optimization may increase the computation time unacceptably. In the above example, however, the added complexity of saving the result would generally be exceeded by the cost of performing the evaluation twice.

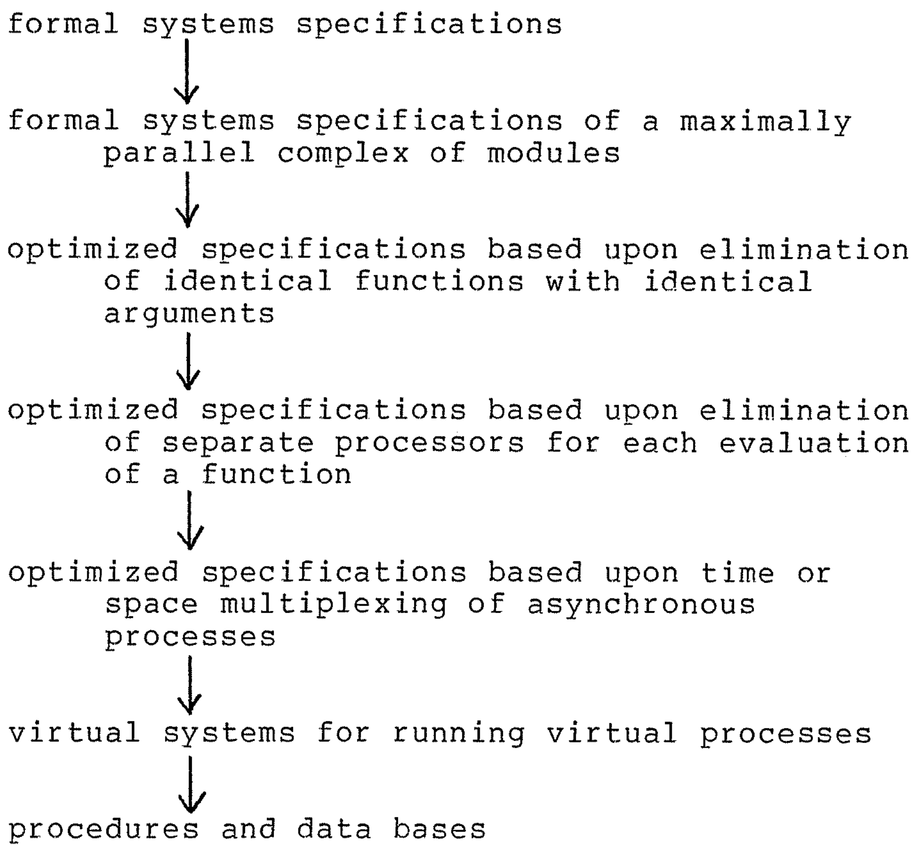

formal systems specifications
formal systems specifications of a maximally parallel complex of modules
optimized specifications based upon elimination of identical functions with identical arguments
optimized specifications based upon elimination of separate processors for each evaluation of a function
optimized specifications based upon time or space multiplexing of asynchronous processes
virtual systems for running virtual processes
procedures and data bases

Figure 17. Steps in an elementary design process.

The same expression which we used above also serves as an example for a second and subsequent type of optimization, namely elimination of separate processors for identical functions with different arguments, as g(a,b) and g(c,d) for the function f above. Here the savings in the number of processors needed for the computation is somewhat offset by the complication introduced in the state space as well as by some loss of parallelism. Presumably this optimization could be carried out mechanically, and the result could be demonstrated, again mechanically, to fulfill the functional requirements. This is an example of how analyses interact with functional requirements and design laws in determining the outcome of a design step.

We can include two subsequent types of optimization in our design strategy. Briefly, one is the storing of results of identical computations (in the object system) in one location for later use (by the object system). This necessitates the use of file managers and library managers in the system, which could be introduced in an automated way as part of the design step, thus avoiding additional responsibility on the part of the designer. The next optimization, potentially the most complex to carry out, is the combination of asynchronous parallel systems into a single system through time multiplexing or space multiplexing. (This type of step is clearly a kind of integration step also. However, integration steps in general need not be optimization steps.) The motivation for this last kind of optimization is obvious when we recall that functional specifications were initially factored into the maximally parallel form. This form is in general an unrealistic model for any but the simplest of systems because of the great expense of implementing such a system. The tasks entailed in automatic methods for accomplishing this system composition, based upon the formalism developed so far, is a subject for further study. Finally, we mention briefly the last two steps of the design process of Figure 17. These are the creation of virtual systems for the optimized system specifications and the creation of procedures and data bases within these virtual systems. These two steps thus complete the entire process of transforming the original functional specifications into procedures operating in virtual address spaces. However, we have not indicated how the procedures and data bases are to be mapped onto the physical hardware, since our design process is concerned with the logical part of requirements specifications.

We now address the question of how user-defined exchange functions are to be treated in the framework of our design schema. We have two main concerns with respect to exchange functions and the associated asynchronous interactions which they determine. One is the detectioon and prevention of logical blockage, i.e., deadlock situations. The other is the possibility of integrating into single systems those interacting systems which have been defined by independent functional specifications. This subject is treated in section 6.5.

The subject of logical blockage is discussed extensively elsewhere in the text, especially in Appendix B, where sufficient but not necessary conditions for prevention of logical blockage are derived with respect to functional specifications. The primary motivation for introducing _allowed_ (see Appendix B) specifications is to avoid the computational complexity of deciding in the general case whether or not a particular functional specification is subject to deadlock. Thus allowed specifications define a significant subset of all unblocked sets of interacting processes. Yet it is not difficult to devise fairly simple examples of specifications which do not introduce deadlock but which on the other hand are not allowed specifications. Since the latter specifications may be of interest to the systems designer we will propose the following set of flexible design tools for dealing with exchange functions.

We assume that a designer wishes to test his functional specifications for the possibility of deadlock. First, if the specifications are found algorithmically to be "allowed" then the task is accomplished. However, if the specifications are not "allowed" then the designer can be queried whether or not the specifications should be tested exhaustively for blockage, a procedure which is feasible only in cases where relatively few exchange functions are involved. Next, if the specifications are found to be unblocked, then the designer may be satisfied or he may ask that the specifications be converted via an equivalence transformation to allowed specifications. On the other hand if the specifications define blocked processes, then the design tools should present to the designer some possible transformation operations for preventing blockage, namely by permitting a set of asynchronous interactions restricted with respect to the interactions associated with the original specifications. The latter transformations are clearly not equivalence transformations, and there may be many possible choices among them which can prevent blockage. Of course, it is possible that the designer may wish to make more comprehensive changes to his functional specifications and check them again for blockage by the design process just described above.

6.5 System Integration

We now introduce a type of integration design step which is possible only in functional specifications which define processes that interact via exchanges. Specifically, we are interested in detecting processes which by virtue of their specifications must interact in locked step fashion as described in the beginning of section 4. (Note that integration steps discussed previously have been intrasystem and that the type now discussed is intersystem.) A necessary and sufficient condition for locked step interaction can be specified very briefly as follows. If we have a set of functional specifications which define a set P of asynchronously interacting processes, then we can consider the set S of all subsets of P which are themselves unblocked. Any group of processes T which for every intersection of pairs of members of S is either wholly contained in that intersection or wholly excluded from that intersection constitutes a set of processes in locked step. Intuitively, no member of T belongs to an unblocked process in S which does not also contain the other members of T. Thus the processes of T must proceed with a fixed ratio of system steps. (Further detail and examples will be provided in a subsequent report.)

It is of interest to note that intersections and unions of members of the set S above may in general be blocked or unblocked. However, in those fuctional specifications for which all exchange interactions are deterministic (as described in section 4) all intersections and unions of members of S are also unblocked and hence in S. Note, however, that this is a very restricted type of interaction. (These topics will also be expanded in a subsequent report.) The main inference which can be drawn from the above remarks is that in general we cannot integrate unblocked sets of processes to form larger unblocked sets of processes but must perform for the larger system the analytical design steps dealing with exchange functions previously discussed in section 6.4.

6.6 Design Process Summary

Our main purpose in this section has been to show that well-defined design steps could minimize the time, cost, and error susceptibility associated with systems design. We have also tried to show that automation of design steps could relieve the systems designer of tedious responsibilities by providing detailed analytical information at each step in the design, of course with the possibility of iterating steps, and by providing feedback on the consequences of design decisions made interactively by the designer. We have seen many parallels between the formal requirements process steps of section 2 and the design process steps of this section. This similarity results from the fact that the design processes which we have discussed are the logical component of the requirements process. We have also mentioned several questions which must be addressed in future research as we attempt to elaborate and formalize the process steps of the design schema further. In particular we shall develop design tools which deal with exchange functions in order to cope with the difficulties of characterizing the possible behaviors of asynchronously interacting processes.

7. CONCLUSIONS

The preliminary study described in this report provides a conceptual framework within which a research program to address many of the critical issues can be developed with a reasonable confidence of success.

7.1 Summary

We have identified a variety of different processes (each with its special constraints and associated design laws and transformations), an initial hierarchy of system properties (each based on aspects of real-time and distributed systems), and a formal functional specification system (meeting the constraints arising from the processes and system properties). A preliminary exploration of how to use these concepts in the design and analysis of distributed real-time systems opened up some new and highly relevant research areas.

7.2 Evaluation

The probability of success in meeting research objectives of this contract has been increased to a satisfactory level, and the plausibility of this approach has been substantially supported. There do not currently appear to be any potentially fatal weaknesses in this approach. It does not address all relevant issues in the design, but those that are addressed are significant and of potentially very high payoff in terms of cost, time, reliability, and testability. There is reason to believe that extension of BMD problems to distributed real-time systems can be brought under intellectual control and effective management.

7.3 Final Report

This final report extends the results reported in a previous special report and presents them in a more formally integrated way. We have not tried to extend our results beyond the requirements of the design processes or to new system properties. We have completed our survey of the issues, critical problems and their potential solutions for the processes and properties introduced in this report. A more global reseach plan for addressing theses issues was presented in section 2.

7.4 Future Work

We plan to continue work on a modified contract beyond that described in this final report. A research proposal for this future work is given in this section.

7.4.1 Introduction

A brief review of the background for the continuation of this work may help.

7.4.1.1 Current Status

We are currently working under Army Research Contract DASG-6Ø-76-C-ØØ8Ø with the Ballistic Missile Defense Advanced Technology Center, Huntsville, Alabama. These problems have been addressed in that contract.

A brief summary of the current status of this work is given below.

- Development process. A discrete process model has been developed.
- Requirements specification. A top-down derivation of some essential requirements on requirements specification has been developed. Some typical and critical steps have been identified as candidates for methodology improvements.
- Formal specifications. A formal functional specification language and interpretation has been developed. This formalism allows specification of asychronously interacting systems and processes at varying levels of abstraction and appears to be a suitable vehicle for developing our methodology.
- Design Process. Some critical steps have been identified and a top level characterization of the process has been obtained. The crucial aspect is "most local" testability of design decisions.
- Critical properties of specifications. In addition to the usual properties of a specification itself, we have explored some additional properties required

of the specified system. The most basic of these is that the systems in the complex do run and complete their process steps and can be shown to do so at the specification level (i.e., the question of whether the specifications really specify a system complex). Preliminary design laws to ensure this behavior and its testability have been developed.

- Top-down methodology development. An approach to the development of a suitable methodology and design science has been explored.

7.4.1.2 Objectives

The long term objectives of this research (see section 1) are reasonably clear and need little interpretation. The major problem lies in quantifying such goals and measuring progress toward them.

The short term objectives are designed to build on our previous results and extend the domain of the specified system properties testable at the specification level, to critical problems identified by the previous work.

7.4.1.2.1 Define DDP Design Theories

The design process that accepts abstract functional specifications and produces DDP process specifiations must be elaborated to provide the basis for the specification of methodology tools to support it. This study must be top down and systematic with respect to the selected properties.

This objective must be met prior to tool development and prototype demonstrations required in the long term objectives.

7.4.1.2.2 Define Critical Real-Time Properties

Performance testing and prediction are a vital part of the development methodology. We must find design laws that will enable us to model, analyse and predict performances from specifications. We must also find design laws that simplify the required real-time performance testing process for DDP systems.

7.4.1.2.3 Conduct Requirements Analysis

We plan to extend current studies of requirements specifications process and its associated methodology, and develop specifications for tools and requirement specifications. The resulting specifications should be suitable for input to the design process discussed above.

7.4.1.2.4 Define Evolutionary Processes

Changes at requirements, design, implementation, and operational levels are inevitable. If we design for change, we may decrease its impact and increase the domain of feasible changes. Evolutionary processes that will support such changes at each of these levels must be designed. Again, the necessary price will be accepting sufficient design laws to allow evolutionary process models to be used.

7.4.1.2.5 Identify Potential BMD Payoffs

Plausible arguments must be developed to support estimates of BMD payoffs. The important impacts of developing methodology on the developments of real-time DDP systems must be identified.

7.4.1.3 Research Requirements

The previous work on contract DASG-60-76-C-0080 will be continued and extended to real-time systems. In each of the areas discussed below, critical issues will be identified and potential solutions developed in the context of the previous work. In each area, the critical comparison with other representative state-of-the-art methodologies will be made, and the potential impact of this work on Ballistic Missile Defense problems will be identified.

7.4.1.3.1 Distributed Data Processing Design

The contractor will develop procedures useful for transforming data processing subsystem requirement specifications into process specifications for a network of virtual, high level, machine-independent systems. The specifications and procedures should be suitable for designers to encode and analyse functional assignments to nodes and to processes within a node.

This design process will include system decomposition and integration steps as well as canonical generation of both control and data structures. Functional analysis and simulation procedures are also required.

7.4.1.3.2 Real-Time Systems

The contractor will identify critical properties for real-time systems that, if present, will decrease required real-time testing and increase the scope of effective testing against requirements. Sufficient conditions on the development process to ensure these critical properties in the resulting design, and corresponding design laws to make them effective will be developed. Tools for applying the required analysis and testing will be specified.

Dynamic models suitable for either analysis or simulation must be developed, and design laws sufficient to make the models applicable must be developed.

7.4.1.3.3 Evolutionary Processes

The contractor will identify critical specification properties that, if present, will limit the impact of evolving requirements or design changes. Sufficient conditions on the development process to ensure these critical properties in the resulting specifications, and the corresponding design laws to make them effective will be developed. Tools for applying the required analysis and testing will be specified.

Meeting this requirement will involve a formalization of such evolutionary processes and a careful structuring of interactions in the evolving system.

7.4.1.3.4 Requirements Analysis

The contractor will assess the impact on the data processing subsystem requirement specification of the properties and conditions developed above. The contractor will specify development guidelines and analysis tools sufficient to ensure these properties and conditions.

Each system property required by the development methodology may generate design laws or guidelines for any previous stage of the development process. Some of these will even have implications on requirement specifications.

7.4.2 Overall Approach

The general approach described by the previous contract reports will be followed in this research work and appears to be a satisfactory basis for this work. Because of the complex nature of this research, details of the approach must be produced, tested, and elaborated during the work. We can identify some of the required tasks discussed below. Undoubtedly others will also be required.

7.4.2.1 Formal Specifications

The formal functional specifications previously developed must be extended as required to support the other tasks. This work will produce a specification language and procedures for analysis and simulation of the specified systems.

Since the formal specifications form the representation medium for encoding requirements and design decisions, we must study equivalence relations as a foundation for optimization of the design process. More relaxed sufficient conditions for algorithmic implication will be developed and extended to characterize more general system complexes. Formal functional simulational procedures must be developed that allow study of the behavior of specified systems at any level of abstraction.

7.4.2.2 Distributed Data Processesing Design

Both static and dynamic models for system decomposition/integration will be developed to support performance impact analysis of proposed design decisions. Procedures for encoding control and data structure design decisions and their organization into a formal automatable methodology and requirements for such tools will be developed.

7.4.2.3 Real-Time Systems

There appear to be three aspects to this task. The first is the analysis of critical reflex paths in a specified system to demonstrate that they are bounded and provide a model of performance coupling with other paths.

The second is to simplify the performance surface (e.g., to ensure that it is convex) and thus simplify real-time testing. The third is to find sufficient design laws to minimize the required real-time testing. We plan to pursue all three possibilities.

7.4.2.4 Evolutionary Processes

This is a relatively unexplored area and the critical issues must be identified. Our formal specification methodology will allow us to define a domain of evolutionary processes. We will then develop sufficient conditions such that the changes can be analyzed, controlled, and automated with minimal and predictable impact on the remainder of specifications. The model for virtual networks involved in DDP design will also be required to define potentially reachable evolved systems.

7.4.2.5 Requirements Analysis

The requirements methodology based on our formal specifications will be elaborated and a small (hand-worked) example will be developed as an illustration and as a source of experience with methodology. Additional analysis and simulation tools will be specified.

7.4.3 Critical Issues

The following are some of the critical issues to be addressed.

7.4.3.1 Distributed Data Processing

- Equivalence. What are equivalence classes? Which members are optimal with respect to performance requirements?
- Distributed data and control models. How suitable are the proposed virtual networks? What canonical structures are sufficient?
- Decomposition/integration models. The decomposition into application and virtual operating systems results in coupling functional paths via resource contention. The virtual (and physical) operating systems must make such contention analysable.
- Functional simulation. What techniques are applicable and how can they be exploited?

7.4.3.2 Real-Time Systems

- Conditions for path flow analysis and prediction for boundedness and performance.
- Resource mapping and contention resolution properties required for simple performance surfaces.
- Interaction conditions that minimize, localize, and simplify real-time testing.

7.4.3.3 Evolutionary Processes

- Definition of domain of evolutionary processes.
- Localization and delegation of design decisions.

- Specification analysis of required process invariances.
- What are required process invariances?

7.3.4 Requirements Analysis

- How to develop example prior to development of methodology tools.
- How to compare with current methodologies.

7.5 Acknowlegements

The author gratefully acknowleges the assistance of many colleagues, former students and present students in developing the concepts in this report. It is not feasible to mention them all here. I do explicitly acknowlege the many valuable discussions and contributions of Prof. Pamela Zave at the University of Maryland. My research assistants at the University of Wisconsin have, of course, been of substantial assistance in preparing this report and are John Compton, Bill Hibbard, Cynthia Hintz, Manfred Klopprogge, James Maki, and Wolfgang Weber.

APPENDIX A - FUNCTIONAL PROCESS SPECIFICATIONS

The functional notation which we use for specifying processes is given by the following series of definitions.

Definition: A <u>value space</u> V is an unspecified primitive set. We use $V_i = V$ for $i = 1,2,\ldots,n$.

Definition: A <u>state component space</u> $\Sigma_i = P(V_i)$ is the power set of V_i, and a <u>state component</u> $\sigma_i \in \Sigma_i$ is a subset of V_i.

Definition: A <u>state space</u> $\Sigma = \overline{\underline{X}}_{i=1}^{n} \Sigma_i$ is a cross product of state component spaces, and a <u>state</u> $\sigma \in \Sigma$ is an n-tuple $(\sigma_1, \sigma_2, \ldots, \sigma_n)$ of state components.

Definition: A <u>process</u> is a pair (Σ, f), where Σ is a state space and f is a <u>(non-deterministic) state successor function</u> which produces a state σ' when applied to a state σ.

f is much like a relation on $\Sigma \times \Sigma$ except its output σ' may depend on interactions with other processes. In the non-interacting case f would be a relation on $\Sigma \times \Sigma$. In either case we need to define some notation for specifying f.

Definition: f is specified by an m-tuple $(g_1, g_2, \ldots, g_m)$ where $g_i = (D_i, R_i, f_i)$ with $D_i, R_i \subseteq \{1,2,\ldots,n\}$ and f_i is a <u>(non-deterministic) component successor function</u> with domain $\overline{\underline{X}}_{k \in D_i} \Sigma_k$ and range $\overline{\underline{X}}_{k \in R_i} \Sigma_k$. f is specified by $(g_1, g_2, \ldots, g_m)$ as follows: $f(\sigma_1,\sigma_2,\ldots,\sigma_n) = (\sigma_1',\sigma_2',\ldots,\sigma_n')$ where $\sigma_i' = \bigcup_{1\leq j\leq m} P_i(f_j((\sigma_K)_{K\varepsilon D_j}))$. $P_i(f_j((\sigma_K)_{K\varepsilon D_j}))$ represents the projection of $f_j((\sigma_K)_{K\varepsilon D_j})$ onto $\sum_i$ if $i \in R_j$, and is the empty set ϕ if $i \notin R_j$.

Definition: A component successor f_i may be specified by an ℓ-tuple $(f_{i1},f_{i2},\ldots,f_{i\ell})$ of <u>value successor functions</u> where f_{ij} has domain $\underset{K\varepsilon D_i}{\overline{X}} V_K$ and range $\underset{K\varepsilon R_i}{\overline{X}} V_K$. f_i is specified as follows: $f_i((\sigma_K)_{K\varepsilon D_i}) = (\sigma_K')_{K\varepsilon R_i}$ where for each $K \in R_i$, $\sigma_K' = \bigcup_{1\leq j\leq \ell} (\bigcup_{Z\varepsilon \underset{S\varepsilon D_i}{\overline{X}} \sigma_S} P_K(f_{ij}(Z)))$. Similar to the previous definition, $P_K(f_{ij}(Z))$ represents the projection of $f_{ij}(Z)$ onto V_K. Note $Z\varepsilon \underset{S\varepsilon D_i}{\overline{X}} \sigma_S$ is a tuple $(Z_S)_{S\varepsilon D_i}$ with $Z_S \in \sigma_S \_ V_S$.

We have defined ways of decomposing a state successor f of a process $(\sum,f)$ into component successors f_i, which are set functions, and if desired into value successors f_{ij}, which are element functions. The f_i and f_{ij} in a specification may be left as primitives or may be decomposed into trees of lower level primitives by operations of composition, subtree selection, and primitive recursion.

Definition: A function f may be specified as a <u>composition</u> $f(x) = h(g_1(x),g_2(x),\ldots,g_k(X))$ where domain f = domain g_i, $1 \leq i \leq k$, range f = range h, and domain $h = \underset{1\leq i\leq k}{\overline{X}}$ range g_i.

This may be applied with non-deterministic functions, and for f in a decomposition tree of an f_i or an f_{ij}.

Definition: A function f may be specified by <u>subtree selection</u> $f(x) = (g_1(x):h_1(x), g_2(x):h_2(x), \ldots, g_{k-1}(x):h_{k-1}(x), h_k(x))$ where domain f = domain g_i = domain h_i, $1 \leq i \leq k$, range f = range h_i, $1 \leq i \leq k$, and $\{\text{true}, \text{false}\}$ = range g_i, $1 \leq i \leq k-1$. This specification models a flow of control which, in terms of if-then-else, is

```
f(x) = if $g_1(x)$ then $h_1(x)$ else
       if $g_2(x)$ then $h_2(x)$ else
       . . .
       if $g_{k-1}(x)$ then $h_{k-1}(x)$ else $h_k(x)$.
```

This may be applied with non-deterministic functions, and for f in a decomposition tree of an f_i or an f_{ij}.

Definition: A function f may be specified by <u>primitive recursion</u> if its domain can be expressed as $S \times D$ where S is ordered by $n: S \to N$, N = the positive integers. Then f is specified as $f(s,y)$ = if $n(S) > 1$ then $h(s,y,f(m(s),y))$ else $g(y)$ where $m: S \to S$, $n(m(s)) = n(S) - 1$ if $n(S) > 1$, range f = range g = range h, domain $g = D$, and domain $h = S \times D \times$ range f. This may be applied with non-deterministic functions if the restriction on m is obeyed, and for f in a decomposition tree of an f_i or an f_{ij} with the proper ordering on component spaces and value spaces.

The idea of the value successor function may be applied more generally. A component successor function may be decomposed by the operations of composition, subtree selection, and primitive recursion. The primitives into which it is so decomposed may themselves be functions whose domain and range are spaces of sets, similar to component successor functions. These primitive functions may themselves be decomposed into value functions. More precisely if g: $\underset{K\varepsilon D}{\overline{X}} S_K \to \underset{K\varepsilon R}{\overline{X}} T_K$ is a function occurring in the decomposition of a component successor, with $S_K = P(V_K)$, $T_K = P(V_K)$ defined as power sets, then g may be decomposed into an ℓ-tuple $(g_1, g_2, \ldots, g_\ell)$ where g_j has domain $\underset{K\varepsilon D}{\overline{X}} V_K$ and range $\underset{K\varepsilon R}{\overline{X}} V_K$. g is decomposed as follows: $g((\sigma_K)_{K\varepsilon D}) = (\sigma'_K)_{K\varepsilon R}$ where for each $K \varepsilon R$, $\sigma'_K = \underset{1 \leq j \leq \ell}{U} \left(\underset{Z\varepsilon \underset{S\varepsilon D}{\overline{X}} \sigma_S}{U} P_K(g_j(Z)) \right)$. P_K is a projection operator as defined previously.

The notation developed for component successors and value successors is rather cumbersome and a clearer notation is useful. For example a state successor function f: $\sum_1 \times \sum_2 \times \sum_3 \to \sum_1 \times \sum_2 \times \sum_3$ may be composed of f_1, f_2, f_3, and f_4 as:

$$f: \sum_1 \times \sum_2 \times \sum_3 \to \sum_1 \times \sum_2 \times \sum_3 \equiv \begin{array}{l} f_1: \sum_1 \times \sum_2 \to \sum_1 \\ f_2: \sum_2 \to \sum_3 \\ f_3: \sum_2 \times \sum_4 \to \sum_1 \times \sum_2 \\ f_4: \sum_3 \to \sum_3 \times \sum_4 \end{array}$$

Following this would be decompositions of the f_i. For the component successor f_1 we could specify value successors f_{11} and f_{12} as:

$$f_1\colon\ \Sigma_1 \times \Sigma_2 \rightarrow \Sigma_1 \equiv \begin{array}{l} f_{11}\colon\ V_1 \times V_2 \rightarrow V_1 \\ f_{12}\colon\ V_1 \times V_2 \rightarrow V_1 \end{array}$$

The operations of composition, subtree selection, and primitive recursion allow us to decompose the f_i and f_{ij} into lower level primitives. These primitives may express any functional relationship which we do not wish to further decompose. To express interaction we have three primitive exchange function types XA, XC and XS. Their evaluations must synchronize with one another, so evaluation control and precedence must be discussed. Evaluating a state successor function f causes each component successor f_i to be evaluated once, which in turn causes value successors f_{ij}, if they were specified, to be evaluated some number of times, possibly zero. In function composition the functions g_i must all be evaluated before the function h is, in order to produce values for h's arguments. In subtree selection some g_i and h_i may not be evaluated, and the order of evaluation of those evaluated is constrained by the if-then-else expression. In primitive recursion the function h is evaluated some number of times, possibly zero, and the order of evaluation is given by the generated composition tree and the if-then-else expressions.

These are the only constraints on evaluation without exchanges. However any evaluation of an exchange must pair with the evaluation of some exchange with the same subscript, according to the following constraints. An XC_i may pair with an XC_i, XA_i, or XS_i. An XA_i may pair with an XC_i or XS_i. An XS_i may pair with an XC_i or XA_i, except that an evaluation of an XS_i may pair with itself. An evaluation of neither an XC_i nor an XA_i may pair with itself. Thus an evaluation of an XC_i or an XA_i must wait for the evaluation of another exchange with the same subscript and of the allowed type, but an XS_i evaluation may pair with itself and evaluate immediately (XS_i is called an immediate exchange). Every exchange has one argument; when evaluations of exchanges are paired, the output value of one exchange is the argument of the other. An XS_i which pairs with itself thus evaluates to its own argument value.

Within a single process specification different instances of the same function may be optimized to be evaluated together if they have identical specifications of their arguments. That is a single evaluation produces a value for all instances indicated; the indication is done by giving the instances equal superscripts. With the same superscript the instances are no longer regarded as distinct, particularly in the case of exchanges. For example if a process specification contains

$$...f(x,g(y))...f(x,g(y))...$$

then these two instances will be evaluated separately, but if we add superscripts

$$\ldots f^{[1]}(x,g(y))\ldots f^{[1]}(x,g(y))\ldots$$

then one evaluation will supply the value of both instances. Notice they have identical arguments, which is necessary. With exchange functions if we consider

$$f(XC_1(A),XC_1(A),XC_1(B))$$

then any of these exchanges could pair with any other but if we add superscripts

$$f(XC_1^{[1]}(A),XC_1^{[1]}(A),XC_1(B))$$

then the first two are considered as one and must pair with $XC_1(B)$; the first two both evaluate to B and $XC_1(B)$ evaluates to A.

APPENDIX B - CONDITIONS ON EXCHANGES

Given an arbitrary specification of a set of interacting processes, it is possible that a computation of these processes may reach a point where several processes cannot proceed, defined as follows:

Definition: Given a functional specification of a set of processes which have gone through a sequence of computation, a process is blocked by an exchange in that process if no matching exchange can be evaluated for any continuation of the computation. This situation, commonly called deadlock, cannot be allowed to develop. We must find sufficient constraints on our functional specifications ensuring that processes are never blocked by exchanges.

Given an arbitrary specification, there is no algorithm for deciding if its processes can ever become blocked. This is because exchange evaluations may be dependent on the values in the process states. Even if we consider a view of computation which is value independent (that is control may flow in any way which is consistent with the definitions of the control structures), the algorithm for deciding if processes can ever become blocked is combinatorially infeasible. However the problem may be attacked by separately considering different types of interactions and their own typical ways of blocking processes. Different types of interactions involve distinctions in intraprocess versus interprocess, XS-XC versus XC-XC versus XC-XA, and exchanges which are always evaluated versus exchanges in subtree selectors versus exchanges in value successor. With regard to the last distinction we will make an assumption of value independence; as long as a process is not blocked it must take each alternative in a subtree selection sometime again, and value successor functions must sometime again have at least one value on which to evaluate. By sometime again we mean that after any point in the computation it must occur at some time in the future.

In order to discuss types of interactions we need the following definition.

Definition: An exchange class is named by an integer index and includes all exchanges in a specification which have that index. XC_i has index i. Examples elsewhere in this report use mnemonic indexes like $XMESSAGE_i$, but here we allow only integers.

Now let us look at a single process P which has a state successor function of the form

$$f(XC_3(XC_2(XC_1())),XC_2(XC_3(XC_1())))$$

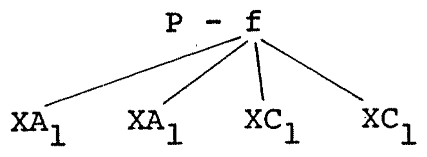
Graphically this is

$$\begin{tikzcd}
& P - f \arrow[dl, dash] \arrow[dr, dash] & \\
XC_3 \arrow[d, dash] & & XC_2 \arrow[d, dash] \\
XC_2 \arrow[d, dash] & & XC_3 \arrow[d, dash] \\
XC_1 & & XC_1
\end{tikzcd}$$

The lowest level functions are evaluated first, the two XC_1's, and they may match with each other. However then one XC_2 and one XC_3 are waiting to evaluate, but since they each must precede the other's possible matching pair, neither can evaluate and process P is blocked. This type of blocking may be avoided with a constraint that if an intraprocess exchange XC_i must precede an intraprocess exchange XC_j then $i < j$. As a part of our divide and conquer strategy for deadlock we also separate intraprocess exchange classes from interprocess exchange classes. That is, if an exchange class has members in more than one process, then within one process that class has exactly one of the following: (a) no members, (b) one XC, (c) one or more XS, (d) one or more XA. Furthermore, intraprocess interactions involving XS and XA exchanges usually cause blocking, so that intraprocess exchange classes contain only XC's, and each process cycle must evaluate an even number of them. An example of intraprocess XS blocking is:

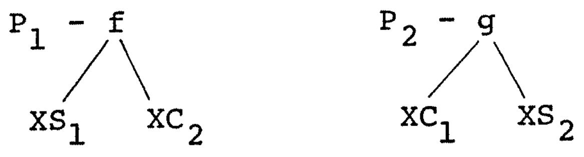
$$\begin{tikzcd}
& P - f \arrow[dl, dash] \arrow[d, dash] \arrow[dr, dash] \arrow[drr, dash] & & \\
XS_1 & XS_1 & XC_1 & XC_1
\end{tikzcd}$$

If one XS matches itself and the other XS matches an XC, the other XC waits forever. Intraprocess XA may block:

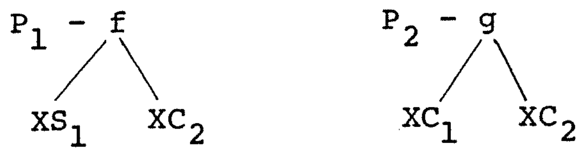


If the XC's match them both XA's wait forever.

Next let us look at interprocess interactions which involve XS and two processes P_1 and P_2:



If both XS_1 and XS_2 match themselves, then both XC_1 and XC_2 wait forever. Antoehr example is:



Here we need to consider a sequence of actions:

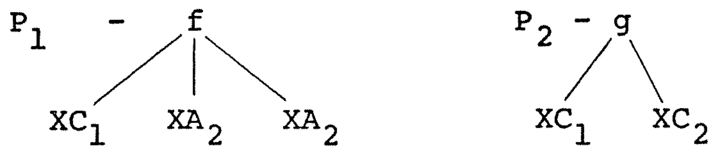
(1) XS_1 matches with itself

(2) XC_2 of P_1 matches XC_2 of P_2 (and P_1 starts another cycle)

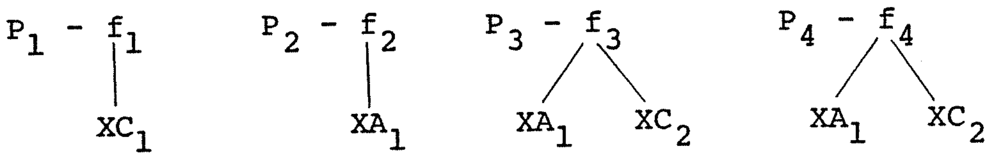
(3) XS_1 matches with itself.

At this point XC_2 of P_1 and XC_1 of P_2 both wait forever. These types of blocking may be avoided by an ordering on the processes of a specification. That is if T is the set of processes and I^+ the positive integers then there is a mapping LEVEL: $T \to I^+$ such that interactions of XS_K in P_1 with XC_K or XA_K in P_2 may exist only if LEVEL(P_1) < LEVEL(P_2) and interactions of exchange classes which have no XS members may exist only between processes at the same LEVEL. Note that this definition of LEVEL makes a distinction between exchange classes which have at least one XS member and classes with no XS members; this distinction is another example of our divide and conquer strategy. This distinction also introduces a hierarchy to a specification; the processes are separated into levels and the levels are ordered to make up the specification.

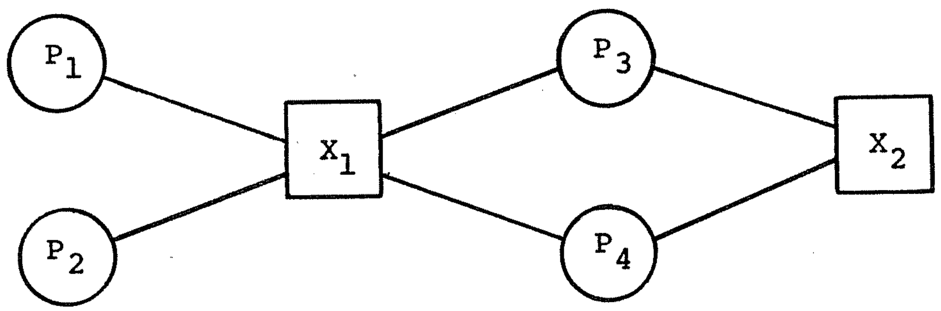
Next we must look at interactions between processes in the same level. These are the interprocess exchange classes which have no XS members. The exchanges here restrict the relative rates of the processes involved and may lead to inconsistent restrictions. An example of this type of blocking is given by:



The X_1 class makes P_1 and P_2 cycle at the same rate but the X_2 class makes P_2 cycle at twice the rate of P_1. We may think of the blockage occurring because processes P_1 and P_2 are connected by two exchange classes, forming a loop. We want to separate interactions involving loops, which may generate rate inconsistencies, from other interactions within levels. To do this, given a specific level, form a graph. Its nodes are processes in that level and interprocess exchange classes which involve processes in that level and which have no XS members. An arc connects a process node and an exchange class node only if that exchange class has a member in that process. Given this graph we separate the processes into <u>cliques</u> as follows; given an exchange class X and two processes P_1 and P_2 both with arcs to X (i.e., containing members of X), processes P_1 and P_2 are in the same clique if there is another path from P_1 to P_2 not containing X. Cliques are the smallest sets of processes consistent with this condition. An example of how this is applied will help:



These four processes are in the same level and assume exchange classes X_1 and X_2 have no members in other processes. There are nodes for the processes and exchange classes and arcs between them as follows:



Now X_2 has arcs to both P_3 and P_4 and there is a path from P_3 to P_4 not containing X_2, namely P_3-X_1-P_4, so P_3 and P_4 are in the same clique. A similar analysis with X_1 and all pairs of processes requires only P_3 and P_4 to be in the same clique, so the cliques are $\{P_1\}$, $\{P_2\}$, and $\{P_3,P_4\}$. Note this corresponds to our idea of loop, as the only loop in the above graph involves P_3 and P_4. Thus each level, as a set of processes, is broken into a disjoint union of cliques. The interactions within cliques involve loops while the intralevel interactions between cliques do not involve loops. Cliques add another element to the hierarchy of process grouping which goes from process to clique to level to whole specification, each properly contained in the next.

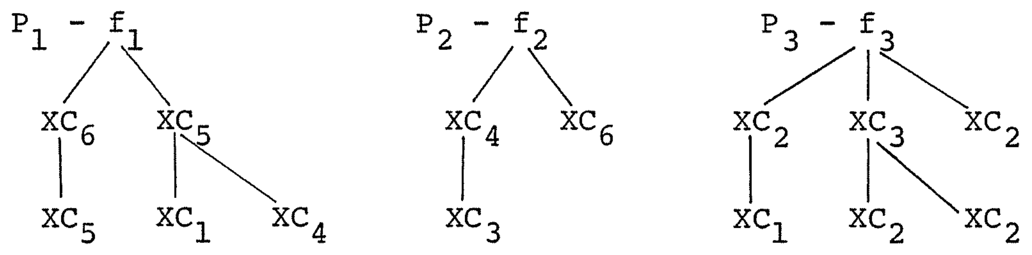
Being loopless, the interactions between cliques do not cause much trouble; there is only one restriction required. If X_K is an exchange class without XS members, that is an intra-level class, and has members in two or more cliques then within a single clique X_K may not have both XC and XA members. This restriction will be needed in the proof of the theorem about nonblocking later. The following example illustrates blocking when the above restriction is not obeyed:

$$\begin{tikzcd}
P_1 \arrow[r, dash] & f_1 \arrow[d, dash] & P_2 \arrow[r, dash] & f_2 \arrow[d, dash] & P_3 \arrow[r, dash] & f_3 \arrow[d, dash] \\
 & XC_2 \arrow[d, dash] & & XC_1 \arrow[d, dash] & & XA_2 \\
 & XC_1 & & XA_2 & &
\end{tikzcd}$$

This specification clearly is blocked. The cliques are $\{P_3\}$ and $\{P_1,P_2\}$. The intralevel interclique exchange class X_2 has an XC_2 in P_1 and an XA_2 in P_2, both in the clique $\{P_1,P_2\}$. In the formal definition later which summarizes all this, the definition of clique is given differently. Here we have shown how to construct the cliques from a specification but later we merely give conditions which cliques must obey. The definitions are equivalent.

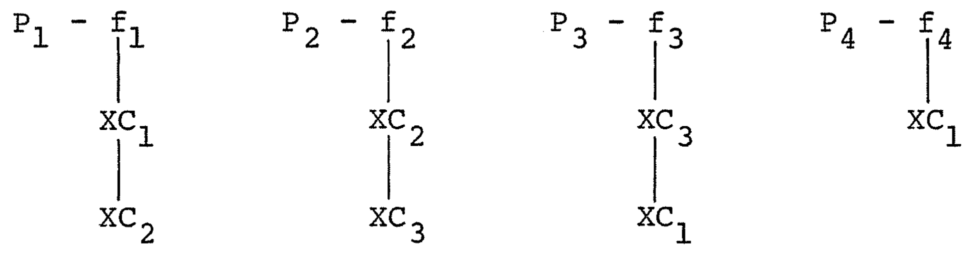
Interactions within cliques include loops and are difficult to analyze. The general analysis is combinatorially infeasible, so we must find conditions on exchange usage within cliques which ensure that blocking cannot occur. We have been separating the types of interactions and considering them separately. With cliques this process is somewhat different; we find different types of cliques which do not block. Specifically so far we have two types of cliques, <u>pair cliques</u> and <u>sparse cliques</u>. With both types intraclique exchange classes may only contain XC exchanges. In a pair clique the idea is that these exchanges may only involve two processes so that these exchanges always match the same partner. In a sparse clique the idea is that any set of N intraclique exchange classes must involve at least N+1 processes. Then, assuming those processes are blocked, two must be blocked by the same exchange class, which is impossible. In a pair clique we also need another condition similar to our condition on intraprocess exchanges. This is because a pair clique behaves like a single process. The condition is that if XC_i and XC_j are intraprocess exchanges or intraclique exchanges and if XC_i must precede XC_j in some process in a pair clique then $i < j$.

The following is an example of a pair clique:



The intraprocess exchange classes X_2 and X_5 each have an even number of members. Each interprocess exchange class X_1, X_3, X_4, and X_6, has one member in each of two processes. And whenever an exchange XC_i preceeds an exchange XC_j we have $i < j$. The clique $\{P_1,P_2,P_3\}$ will behave much like a single process.

The following is an example of a sparse clique:



Here each exchange class has members in at least two processes, each pair of classes involves at least three processes, and the three classes X_1, X_2, and X_3 involve all four processes. The processes cannot become blocked.

The strange kind of trouble which may occur in a clique is shown by an example, which is in a sense the simplest clique which is neither a pair clique nor a sparse clique:

$$\begin{tikzcd}
P_1 \arrow[r, dash] & f_1 \arrow[dl, dash] \arrow[dr, dash] & \\
XC_1 & & XC_2
\end{tikzcd}
\qquad
\begin{tikzcd}
P_2 \arrow[r, dash] & f_2 \arrow[dl, dash] \arrow[dr, dash] & \\
XC_2 & & XC_3
\end{tikzcd}
\qquad
\begin{tikzcd}
P_3 \arrow[r, dash] & f_3 \arrow[dl, dash] \arrow[d, dash] \arrow[dr, dash] & \\
XC_1 & XC_2 & XC_3
\end{tikzcd}$$

Consider this sequence of actions:

(1) XC_2 in P_1 matches XC_2 in P_2

(2) XC_1 in P_1 matches XC_1 in P_3 (P_1 cycles)

(3) XC_3 in P_2 matches XC_3 in P_3 (P_2 cycles)

(4) XC_2 in P_1 matches XC_2 in P_2

Now P_1, P_2, and P_3 are blocked by XC_1, XC_3, and XC_2 respectively.

The result of all this has been to separate interactions into five distinct types and to find conditions on each type which ensure that blocking cannot occur. Each exchange class belongs to one and only one type. The types, along with the names they have in the formal definition below, are

(1) E_1 - intraprocess classes.

(2) E_2 - interlevel classes.

(3) E_3 - intralevel interclique classes.

(4) E_4 and E_5 - intraclique interprocess classes; E_4 are in pair cliques, E_5 are in sparse cliques.

In dealing with each type of exchange one general idea has motivated the restrictions we put on the exchange classes; that is the idea of an order relation. The intraprocess exchange classes are ordered according to their precedence. The exchange classes involving XS interactions set up an ordering of processes into levels. Within levels the interclique exchange classes may not form any loops; essentially this is a partial ordering of the cliques. And within pair cliques the intraclique classes and the intraprocess classes must be ordered together according to precedence. This all bears a relation to the more common notion of ordering resources in a deadlock avoidance scheme.

We can now combine our restrictions on exchanges in a formal definition.

Definition: A specification of interacting processes is an allowed specification if the set E of exchange classes in the specification may be broken into a disjoint union $E = E_1 \cup E_2 \cup E_3 \cup E_4 \cup E_5$ such that the following conditions are obeyed:

(1) The specification contains the set T of processes and the set E of exchange classes.

(2) As long as a process is not blocked it must take each alternatives in a subtree selection sometime again, and value successor functions must sometime again have at least one value on which to evaluate.

(3) If $K \varepsilon E_1$, K is an intraprocess class and has members in only one process. All its members are XC_K, and

each process cycle must evaluate an even number of distinct members in the process, independent of which subtree selections are made. None of K's members may occur in value successor functions or in primitive recursion specifications. If $K, \ell \in E_1$ are in the same process and if some XC_K is constrained to precede some XC_ℓ, then $K < \ell$.

(4) If $K \in E - E_1$, K is an interprocess exchange class and has members in at least two processes. For each process $P \in T$ in which K has members, exactly one of the following occurs

(a) P contains exactly one XC_K

(b) P contains one or more XA_K

(c) P contains one or more XS_K.

No XC_K may occur in a value successor function or in a primitive recursion specification.

(5) $K \in E_2$ if K has an XS_K member. There is a mapping LEVEL: $T \rightarrow I^+$, I^+ = the positive integers, such that

(a) if there is $K \in E_2$ with a member XS_K in P_1 and a member XC_K or XA_K in P_2 then $\text{LEVEL}(P_1) < \text{LEVEL}(P_2)$.

(b) if there is $K \in E_3 \cup E_4 \cup E_5$ with a member XC_K or XA_K in P_1 and a member XC_K or XA_K in P_2, then $\text{LEVEL}(P_1) = \text{LEVEL}(P_2)$.

Note that a class in $E_3 \cup E_4 \cup E_5$ may have members only in processes with the same LEVEL.

(6) If $K \varepsilon E_3$, K is an interclique class and contains at least one XC_K, no XS_K, and zero or more XA_K. Let $T_\ell = \{P\varepsilon T\colon\ \text{LEVEL}(P) = \ell\}$. T_ℓ is a disjoint union of cliques $T_{\ell j}$, $T_\ell = \cup_{j=1}^{m} T_{\ell j}$ where m depends on ℓ. In a clique $T_{\ell j}$ there may not be both XC_K and XA_K. K must have members in at least two cliques. There may not be loops of interclique classes. That is, there may not be cliques $Q_1, Q_2, \ldots, Q_n, Q_{n+1}$, where $n \geq 2$ and $Q_1 = Q_{n+1}$, and classes $K_1, K_2, \ldots, K_n \varepsilon E_3$ such that K_i has members in Q_i and Q_{i+1} for $i = 1, \ldots, n$. A clique is either a sparse clique or a pair clique.

(7) $K \varepsilon E_4$ has one member XC_K in each of two processes, both in the same pair clique. The XC_K may not occur in subtree selection. If $K, \ell \varepsilon E_1 \cup E_4$ and if some XC_K is constrained to procede some XC_ℓ, then $k < \ell$.

(8) $K \varepsilon E_5$ has members XC_K in all processes in the same sparse clique Q. If there are distinct $P_1, P_2, \ldots, P_m \varepsilon Q$ and distinct $K_1, K_2, \ldots, K_m \varepsilon E_5$ such that K_i has a member in P_i for $i = 1, \ldots, m$, then there is a K_i, $1 \leq i \leq m$, and $P \varepsilon Q - \{P_1, P_2, \ldots, P_m\}$ such that K_i has a member in P.

The definition of an allowed specification seems fairly complex. However the idea is to break a specification into a hierarchy of process, clique, and level, and to put conditions on an exchange interaction that depend on its place in the hierarchy. Thus E_1 is intraprocess, E_2 is interlevel, E_3 is intralevel intraclique, E_4 and E_5 are intraclique interprocess. Furthermore, the definition of allowed specification may be expanded in the future by expanding the hierarchy or by changing the conditions on a type of interaction. In particular it will probably be fruitful to add more types of cliques in addition to pair cliques and sparse cliques. Figure 1 illustrates the hierarchical system. Exchange classes are identified by lines connecting possible exchange matchings. State successor functions are assumed to be consistent with an allowed specification. Note Q1 and Q2 are pair cliques and Q3 is a sparse clique. Cliques Q4 to Q8 are single processes.

The proof of the theorem below has the following outline:

I) assume an allowed specification is blocked, and may not be further blocked

II) in the lowest blocked level no process is blocked by an exchange in an E_2 class

III) in that level every blocked process is blocked by an exchange in an $E_3 \cup E_4 \cup E_5$ class

IV) in that level every clique with a blocked process has a process blocked by an exchange in an E_3 class; proved first for sparse cliques then for pair cliques.

V) in that level each E_3 class with an exchange blocking a process must have all its members in cliques with a blocked process

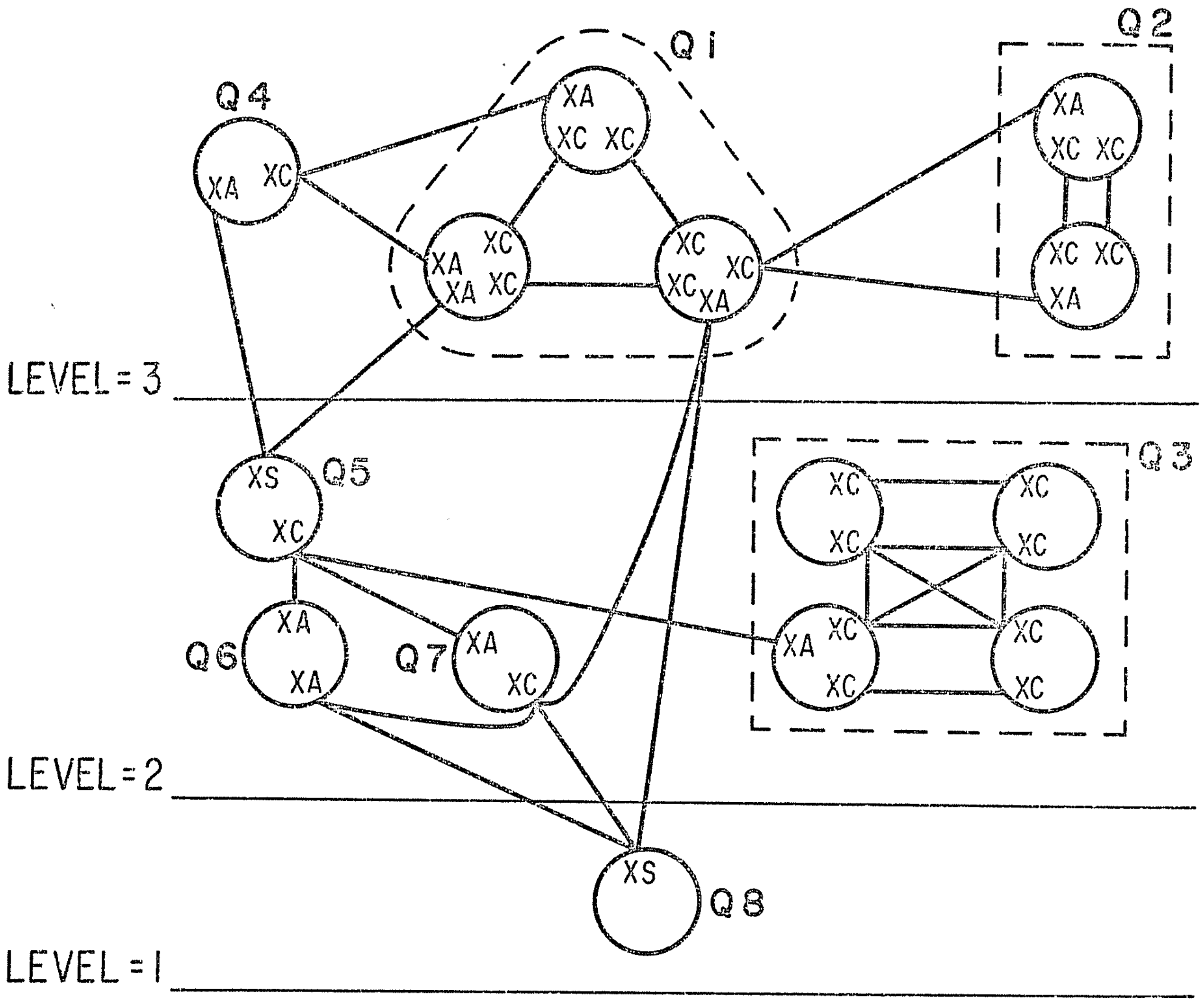


Figure 1

Circles are processes, cliques are labeled Q1 to Q8, and levels are separated by horizontal lines. Lines representing exchange class interactions are labeled by their type E_2 to E_5. No intraprocess classes (E_1) are shown.

VI) the looplessness of E_3 connections between cliques makes the above impossible, so by contradiction the allowed specification cannot be blocked.

The labels of the outline correspond to the labels of paragraphs of the proof.

Theorem: In a computation of an allowed specification, no process can become blocked by exchanges.

Proof:

I) Assume that we have an allowed specification with processes T and exchange classes E which is blocked. Let $B \subseteq T$ be the set of blocked processes. We may assume that no process in T-B can become blocked (B is maximal) and that no further evaluation is possible in any process in B. If either of these assumptions is false, continue running the computation until it is true. Let $\ell = \min\{\text{LEVEL}(P): P \in B\}$ and $M = \{P \in B: \text{LEVEL}(P) = \ell\}$. We say a process $P \in M$ is blocked at an exchange X_K (XC, XA, or XS) if a possible next evaluation for P is X_K and there can never be a match for X_K.

II) If $K \in E_2$ and $P \in M$, then P cannot be blocked at any X_K. Clearly P cannot be blocked at any XS_K, which could match with itself. If P is blocked at an XC_K or XA_K, then by condition (5) there is an XS_K at a lower level in some process P', the minimality of ℓ implies P' must not be blocked, and by condition (2) P' must evaluate XS_K sometime to match with the given XC_K or XA_K in P. Thus $P \varepsilon M$ cannot be blocked by any X_K where $K \varepsilon E_2$, so $P \varepsilon M$ are only blocked by X_K, $K \varepsilon E_1 \cup E_3 \cup E_4 \cup E_5$.

III) If some $P \varepsilon M$ is only blocked by XC_K for which $K \varepsilon E_1$, then let m be the minimum index for which some XC_m blocks P. If any XC_m is constrained to antecede some XC_K blocking P, then $K < m$ by condition (3), but m is minimal so all the XC_m either have been evaluated or are blocking P. But there are an even number of distinct XC_m by condition (3), so there must be an even number blocking P and they can evaluate. Thus each $P \varepsilon M$ is blocked by at least one X_K with $K \varepsilon E_3 \cup E_4 \cup E_5$.

IV) Say Q is a clique such that $M \cap Q$ is not empty. Let $R = M \cap Q$, the processes of Q which are blocked. Now assume that $P \varepsilon R$ and P blocked by X_K implies $K \notin E_3$, that is no processes of R are blocked by interclique exchanges. We will show a contradiction. If Q is sparse clique then our assumption implies each $P \varepsilon R$ is blocked by at least one XC_K with $K \varepsilon E_5$. Say $R = \{P_1, P_2, \ldots, P_n\}$. If P_i and P_j, $i \neq j$, are both blocked by the same XC_K, then they can match and are not blocked. So there are distinct $K_1, K_2, \ldots, K_n \varepsilon E_5$ such that P_i is blocked by XC_{K_i}, and by condition (8) there is some K_i and $P \varepsilon Q - R$ such that K_i has a member in P. But P is not blocked so will, by condition (2), evaluate XC_{K_i} sometime which can match with XC_{K_i} in P_i. Thus P_i is not blocked, a contradiction. Now if Q is a pair clique, then our assumption implies each $P \varepsilon R$ is blocked by at least one XC_K with $K \varepsilon E_4$. Let

$F = \{K \in E_4:$ K has a member in $Q\}$. $K \in F$ has one member in each of two processes and those two XC_K's must always match each other. Let d_K = the number of times the XC_K's have been matched. Let $d = \min\{d_K: K \in F\}$, and let $m = \min\{K \in F: d_K = d\}$. That is XC_m is one with minimal d_m, and among those has minimal m. We can assume that those $P \in Q - R$ have computed enough so that m has its members in two blocked processes, P_1 and P_2. Each of P_1 and P_2 is blocked by at least one XC_K, $K \in F$. Now if XC_m has already been evaluated in the current cycle of P_1, then the XC_K blocking P_1 has been evaluated one less time and $d_K = d_m - 1$ against our definition of m. So XC_K is unevaluated in the current cycle of P_1 and likewise P_2. Also if XC_K, $K \in F$, is unevaluated in P_1 or P_2, then $d_K = d_m$ so $m < K$ by the definition of m. Now by our assumption at the beginning of this paragraph, P_1 or P_2 blocked by $X_K \Rightarrow K \in E_1 \cup E_4$. Thus if XC_m does not block P_1 (or P_2), then XC_m must be preceded in P_1 (or P_2) by some XC_K, $K \in E_1 \cup E_4$, which implies $K < m$ by condition (7). However XC_K unevaluated in P_1 (or P_2) and $K \in E_4$ imply, by above remarks, that $K \in E_1$. So if XC_m does not block P_1 (or P_2), then it is preceded by some XC_K, $K \in E_1$, which blocks P_1 (or P_2). Say XC_j is the intraprocess exchange of smallest index which blocks P_1 (or P_2). But there are no XC_i, $i \notin E_1 \cup E_4$, blocking P_1 (or P_2) and no XC_i, $i \in E_4$ and $i < j \leq K < m$, blocking P_1 (or P_2), so XC_j must be able to match with another XC_j in P_1 (or P_2), as the process has an even number of distinct XC_j. This is a contradiction and thus P_1 and P_2 are blocked by XC_m, in which case the XC_m's may match with each other, a contradiction. Thus in this paragraph we have a contradiction for Q both a sparse clique and a pair clique, and our assumption that $P \varepsilon R$ and P blocked by X_K implies $K \notin E_3$ must be false. So for each clique Q with $R = Q \cap M \neq \phi$ there is a process for $P \varepsilon R$ which is blocked by some X_K, $K \varepsilon E_3$.

V) Let $Q_1, \ldots, Q_n$ be the cliques for which $Q_i \cap M \neq \phi$. A Q_i has at least one process, which is blocked by at least one member of K_i where $K_i \varepsilon E_3$ is an interclique exchange class. Now K_i cannot have any members in a clique Q such that $Q \cap M = \phi$, since then some $P \varepsilon Q$, not blocked, would contain an X_{K_i}. If it is an XC_{K_i} it must sometime be evaluated and may match with X_{K_i} in Q_i. If it is XA_{K_i}, it must sometime be evaluated and find a matching XC_{K_i} in another unblocked process, and that XC_{K_i} may match with X_{K_i} in Q_i. Thus each class K_i has members only in the Q_j, $j = 1,2,\ldots,n$.

VI) Now let $K_1, K_2, \ldots, K_m$ be those $K \varepsilon E_3$ with members only in the Q_j, $j = 1,2,\ldots,n$. Say K_i has members in a_i distinct Q_j. Then, because of the no loop part of condition (6), it can be shown that

$$n \geq \sum_{i=1}^{m} a_i - m + 1.$$

Also say for K_i there are b_i different Q_j which contain a process blocked by some member of K_i. Then, since each Q_j has at least one process blocked by some member of a K_i, we get

$$n \leq \sum_{i=1}^{m} b_i .$$

Combining our two results here, we get

$$m - 1 \geq \sum_{i=1}^{m} (a_i - b_i) .$$

Now by their definition $a_i - b_i$ is a non-negative integer, so for some j, $1 \leq j \leq m$, $a_j - b_j = 0$. That is each clique containing a member of K_j has a process which is blocked by a member of K_j. Now since no clique contains both XC_{K_j} and XA_{K_j}; K_j has members in at least two cliques, and K_j has at least one clique with XC_{K_j} [all by condition (6)], there must be two processes in different cliques which are blocked by X_{K_j}'s that can match. This contradicts our assumption that there were processes which were blocked and proves the theorem. ■

Almost all the statements in the definition of allowed specification are needed for this theorem although sometimes their use in the proof was implicit.

If the definition of allowed specification is expanded in the ways mentioned earlier, it is fairly clear where changes must be made in the proof. It should be possible to use these ideas in the analysis of boundedness for real-time requirements. Also the hierarchical structure can give an orderly way to map a specification into a more developed specification, where for example processes are mapped into cliques, cliques into sets of cliques or levels, and levels into sets of levels. It is easy to see ways of mapping processes into pair cliques and some non-deterministic processes into sparse cliques. There is great potential for developing this work.

APPENDIX C - VIRTUAL NETWORKS AND OPERATING SYSTEMS

The decomposition of a distributed data processing system into application systems, virtual operating systems, physical operating systems, and hardware systems seems essential to our developmental processes. A great deal of previous work on this problem has strengthened this belief. An introduction to some of this work is given in the draft manuscript included in this appendix. Further details may be found in [Kr 73] and [Co 75].

The work reported here was prior to the developmental of the present formal functional specifications and has not been reconsidered in this new context. It does serve to explore the virtual system issues, and so is included here.

[Kr 73] Kramer, John F., A General Structure for Uncooperative Processes Distributed over a System Network. Ph.D. Thesis, University of Wisconsin, Madison, 1973.

[Co 75] Cowan, George, Jr., Management of Resources in a Potentially Hostile Environment (Logical and Physical). Ph.D. Thesis, University of Wisconsin, Madison, 1975.

THE ARCHITECTURE OF A MACHINE INDEPENDENT NETWORK OPERATING SYSTEM FOR HIERARCHIAL DELEGATION OF AUTHORITY

JOHN F. KRAMER, JR.
United States Navy

and

DONALD R. FITZWATER
University of Wisconsin, Madison

ABSTRACT: Networks of digital computer systems are in use today and there is little doubt that their numbers will continue to grow. If these endeavors are to avoid the traps that befell many of the third generation computer operating systems, a practical solution to the problems of system manageability, application generality and process portability must be found. This article presents a set of postulated design constraints, and a network design derived from them, which is a practical solution to these problems. Through a clear factorization of the management responsibilities involved in a network, the design presented is able to permit both the implementers and the users of the network to optimize the management of their own resources with respect to their own criteria.

Key Words and Phrases: network, system, operating system, portability, processes, processor, cooperative processes, process communication.

This research was supported in part by the United States Navy under the Doctoral Study Program. Authors' addresses: John F. Kramer,

Jr., 3-M Officer, Nimitz (CVAN 68) Precommissioning Unit, Newport News, Va. 23607; Donald R. Fitzwater, Department of Computer Sciences, 1210 West Dayton, University of Wisconsin, Madison, Wi 53706.

OUTLINE

1. Introduction---management breakdown
    A. Centralized
    B. Decentralized
    C. Resource factorization
2. Postulated Design Constraints
    A. Network
    B. Responsibility
    C. Authority
    D. Delegation
    E. Modification
3. Systems Structures
    A. Level of Interpretation
    B. Digital Systems
    C. System Processes
    D. Network Processes
4. System Interactions
    A. Phase One
    B. Phase Two
    C. Phase Three
    D. Message Delivery
5. Network Processes
    A. Network Process State

1. Interface Buffer
2. Control Points
3. Expression States
4. Environment
5. Intra-process Transformation

B. Network System Language

1. GP
2. Roles
3. Type Conversions
4. Multi-Lingual

C. Process Tree

1. Conflict Resolution
2. Unique Authority
3. Birth and Death

D. Process Movement

1. Need For
2. Transmission
3. Reliability
4. Translation

6. Resource Management

A. Resources

B. Design Requirements

C. AO

D. Control of Interactions

7. Operating Systems
    A. Implementation
        1. Goals
        2. Machine Dependent
        3. Language Effective
        4. Cooperation
    B. Network
        1. Goals
        2. Machine Independent
        3. Language for Application Independent of Machine
        4. Non-cooperation
        5. Authority---properly nested
    C. Common Subprocessor
8. User Freedoms
    A. Designate Process Control
    B. Standard Interface
    C. Complex Process Structures
    D. Control Point
    E. Modification and Evolution

1. Introduction

The emergence of computer networks has underscored the problems associated with the design of general purpose computer systems. Contemporary operating systems are <u>prima facie</u> evidence that in spite of nearly prohibitive investments, operating systems rarely remain in an error free state, rarely are exploited easily by users and rarely free the users from having to re-develop their application software to adapt a new computer. The enormous costs of systems today are caused in part by merely elaborating on the concept of centralized management of resources in a master/slave mode with little interaction between "slaves." This design concept was developed, and worked well, for early batch processing systems. In order to prevent chaotic conflicts between users due to hardware resource demands, operating system designers were forced to usurp most resource management decisions and become "masters."

If users are not allowed to manage resources to fit their diverse needs then the operating systems must attempt to do so for them. A proliferation of options and compromises emerge which rarely satisfy the knowledgeable user and significantly constrain applications. A network only enlarges the community of interacting users and points up the facility of continuing this <u>ad hoc</u> trend. It is not realistic to expect operating system designers to continue to make all of the management decisions in an efficient fashion while the diversity and complexity of user demands continues to grow.

In conventional systems, resources are usually managed by isolating each job or run and allowing it to interact solely with the operating system. Since only the processes of the operating system can directly interact, with sufficient care they can be managed to behave cooperatively with respect to resource demands. As soon as users are permitted to employ interacting processes using shared information, such cooperation can no longer be assumed or guaranteed. Processes may become uncooperative through malice, error, or lack of facilities for cooperation. Much of the capacity of systems today is spent managing user-specified, unreliable processes. This is the "last straw" which causes conventional operating system designs to bog down. The work reported here is part of a detailed design study found in (8). This paper describes primarily the nature of the resulting network. Only suggestions of the justifications for the decisions made are presented here.

Decentralized management capabilities must be provided in a viable general purpose network so that each involved manager can exercise sufficient authority to meet that manager's goal. The partitioning of overall management responsibilities must be carefully done so as to prevent insoluble managerial conflicts. A "top down" design technique as used for this network design can guarantee such a partitioning with minimal constraints on the policies of each manager. We can clearly distinguish three kinds of management:

1) Implementation managers are responsible for the operation of a physical node in the network. This includes the support

of virtual systems in a virtual network and the mapping of virtual resources onto physical resources via physical "operating system" processes. Aside from physical communication protocols, each physical node may be designed, implemented, and managed independently. Management goals might be maximum investment returns or node utilization.

2) Network managers are responsible for the formal specification and evolution of the virtual network as well as the initial virtual network operating processes. The virtual network serves as the interface between the implementation managers and the virtual process managers. Management goals include the maximal delegation of responsibility (and correspondingly factored authority) to the other managers.

3) Process managers are responsible for the specification and operations of the virtual processes via virtual system programs. Management goals include the creation of suitable virtual "operating system" processes as well as "application system" processes.

We are not here concerned with implementation management decisions, and will use unqualified "network," "system," and "process" to refer to their virtual counterparts.

The recognition of these three kinds of management is required for effective management decentralization. We also obtain a substantial amount of freedom to exploit changes in implementation technology while preserving our investment in application processes. As an additional benefit, our application processes can become portable over the network since they can be expressed in terms of (virtual) network programs. These programs may be interpreted by hardware, micro programs, or normal programs, thus playing the role of machine independent, network "job control" programs. The compilation of subsets of the network system language to machine dependent "typed values" (with virtually inaccessible representations) will allow current jobs (e.g., Fortran programs) to run with normal efficiency, while allowing inter-process interactions over the entire network when desired. We are here primarily describing the resulting network design itself although much more could be said about implementation and application designs.

The network design itself is hierarchically structured with corresponding processor and process design decisions. For example, a processor component may require a particular process state component structure. Equally important, the process of network design is also hierarchically structured, in a "top down" manner. Each level in the design hierarchy has the goal of making only necessary (or non-controversial) design decisions so as to defer to later design levels all other decisions. This technique results in a cleanly structured network that delegates maximum freedom (through invariant network properties) for other managers to do their job.

2. Postulated Design Constraints

In view of the previous arguments, and others as detailed in (8), we postulate the following set of network design constraints to form a basis for further design decisions. Many other useful constraints may be derived from these but there is not space here to discuss them.

A. Network Communication

"The designed network must consist of a set of interacting, bounded, programmable, digital computer systems with well-defined processes, and be open to communication with foreign systems having undefined processes."

B. Responsibility factorization

"Responsibility for meeting the postulated design constraints and any design decisions must be factored between the network, implementation, and process designers."

C. Authority

"Each designer must have sufficient authority to define the effects, on the processes running in the network, of the interactions for which that designer has been delegated responsibility."

D. Delegation

"Although processes must be permitted to delegate to other processes their authority to control process interactions, the delegating process always retains the responsibility for that interaction and the authority to control the process to which it was delegated."

E. Modification

"The network definition must provide for modification of its definition. The design must provide sufficient constraints to be able to delegate safely to process managers all modification decisions and authority to control the effects of such decisions on the processes."

3. System Structures

In addition to not depending on the hardware/software of today, we must formally define our system structures so that each implementer of a node in the network can test the implementation (i.e., there is at least a standard, correct specification). The logical structures which make up the network described here are formally defined in (8) using the technique defined in (6). This technique involes an effective network specification which allows a test of hypothesis about specified system behavior. For the purposes of this paper we will use informal, hopefully intuitive, notions of these system structures.

In general it is solely a matter of interpretation which determines whether a set of information physically represented in a computer system is data, program, processor, or system and is not an intrinsic property of the information itself. Since we must first choose a level and consistently describe structures from that level we must provide a few definitions. A natural choice for this paper is the network definition level since this serves as the partitioning interface between network users and implementation programmers. For the postulated design constraints presented above to be effective, a general model of digital systems and processes must be defined.

A digital system is composed of two parts, _a system processor_ and a _system state_. A system processor is invariant to the processes which it interprets and contains a set of operators on the system state. These operators are applied by the system processor to define a new system state or to construct messages for transmission to other system states in a set of digital systems. The execution of a system processor occurs in two distinct phases. Given an initial system state, the processor evaluates all operators which apply and produces a new local system state. Each operator can be executed in parallel and without side effects on the other operators being executed. The local system state is then updated by any messages from other system processors and the local interpretation cycle begins again. This sequence is called a _system step_ and transforms the system state into a successor state. This discrete sequence of states is called _a digital computation_. A set of digital computations represents _a digital process_. The interpretation cycle of each system in a set of interacting digital systems is independent and asynchronous. All inter-system communication will be defined in terms of asynchronous message transmission with any desired synchronization explicitly carried out by the cooperating digital processes. Messages will be received, after a finite delay, in the same order as they were transmitted by a given system. We will define some particular digital systems as network systems, and a set of such systems as a network. Note that such sets of digital systems could model ordinary computers at several levels of abstraction from one (quite abstracted) system to a network of "flip flop" and "gate" digital systems. There is a universal simulator of such sets of digital systems. Thus the system specification is already in a form that can be studied and tested. We do not require physical implementation constraints on how many networks or network systems may be implemented on a single physical system or on how many physical systems are used for a network or network system.

Our network system states are defined as a set of elements. Each element can be interpreted as the state of a synchronously interacting process component of the overall system process. All system state information and all system computations are based on these system process elements. We will constrain our network system operators so that at most one operator will modify a given system state element in a given interpretation step. This not only prevents true race conditions from arising but also avoids the thorny question of how to merge several modifications of a given system process element into a well defined successor element. With this design constraint we are free to define our operators as any total function of the corresponding state elements, producing a new state element.

Although we wish our system processors to be well-defined with respect to their operations on the system state, we desire also to permit them to communicate with systems outside of the formally defined networks which are not well-defined. Such systems are called foreign systems and are not constrained in their internal structure by the network designers except for communication conventions they must use if they wish to interact with a network systems. Such systems can be thought of as sources and sinks for messages. Certain elements of the system state are housekeeping processes that manage inter- and intra-system interactions. These elements contain system state information and are neither explicitly transformed by application programs nor transportable between systems in the network. Those elements of the state that are explicitly programmed, transformed, and moved from system to system will be called network (i.e., user) processes. A network process step consists of a cycle of three system process steps or phases. Two of the system phases are used for housekeeping functions and will be described later. Operators applied during phase two of a network process step will be called system subprocessors and their effect is local to that network process state. The complexity of a given network process step is therefore defined by the corresponding subprocessor. The process state defines the current address space of the process.

It is not feasible to present here the detailed arguments and justifications for this network design. We will refer the interested reader to (1) and subsequent reports in preparation. We will attempt to describe the major network system structures in the following section. Figure 1 is an informal characterization of one network system.

In review, a foreign system is not characterized (or constrained) except as a source or destination of messages. A network system is composed of a system processor and a system state. A system processor is composed of a set of operators, some of which are subprocessors. The system state is composed of system housekeeping elements and system elements interpreted as network (user) process states. A network consists of an inter-communicating set of such systems and foreign systems.

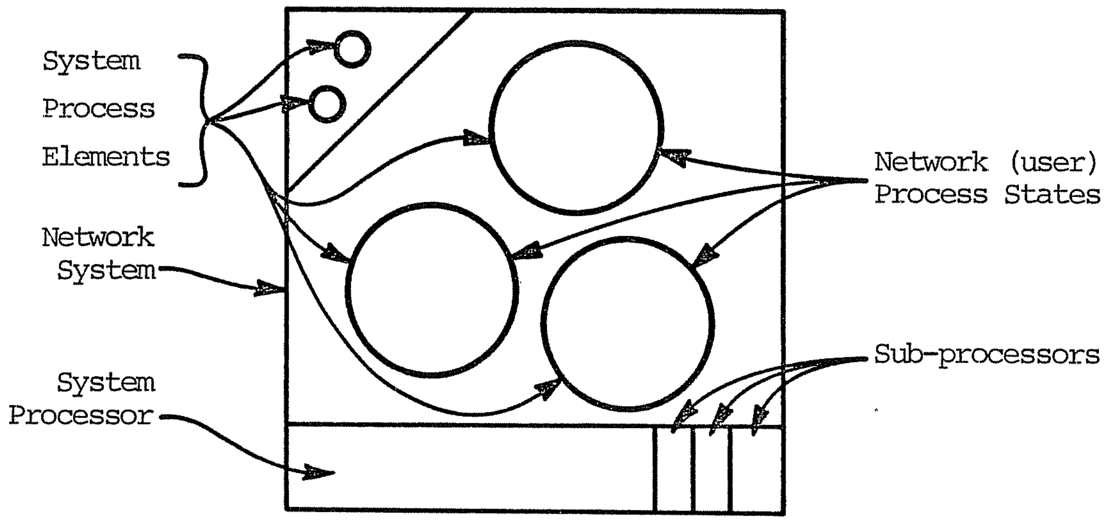


Figure 1: An informal characterization of a network system.

4. Network System Interactions

Inter-system interaction mechanisms are inherently non-local in nature, which means that the responsibility for their design falls to the network designers. The mechanism selected here is quite similar to the notion of a hierarchical interrupt system. In phase one the system processor determines which sub-processors (if any) should be applied to each network process state in that system and delivers the appropriate messages (if any) to an interface buffer in each network process state. In phase two the system processor applies the selected sub-processors to the selected network process states thus causing them to undergo a network process step. In phase three the system processor performs the non-local services requested (if any) by messages left in the interface buffer by a sub-processor. Non-local operations are those which have side effects external to a network process state. Messages may be received by the system processor in all phases and are buffered until their delivery in some subsequent system phase one.

The system processor must be able to recognize, in any network process state, which sub-processor to apply in phase two and each sub-processor must be able to recognize its own state information component in that network process state. For these purposes, each network process state will contain one or more objects called control points. Although control points are used for a variety of purposes, only those aspects affecting sub-processor application will be described here. A control point roughly corresponds to an interrupt level in conventional systems.

All control points in a network process state are ordered by priority within that state. Each control point has an ordered set of channel terminations and buffers associated with it in the system state elements for the receipt of messages (not just an interrupt bit). Control point buffers may be armed or disarmed, and if disarmed, do not accept new messages. Control points may be enabled or disabled and active or inactive. When disabled, no further processing of pending messages in the control point buffers occurs. While active a control point is a candidate to have a subprocessor allocated to it and further messages remain pending in the control point buffers. When a control is inactive but enabled and has a message pending, it is considered as a candidate for message delivery, conversion to active state and subsequent subprocessor application. The highest priority control point candidate present in a network process state, will be made active and the pending message will be placed in the process interface buffer. Two methods of message delivery are provided depending on a message-handling status bit. Either the first message in the ordered control point buffers is delivered or an ordered concatenation of all messages for that control point is delivered. Message overwrite occurs if a message arrives at an armed buffer with a previously undelivered message in it. This may be avoided if desired by programming processes so they use appropriate synchronizing messages

During phase two the appropriate subprocessors are applied to network process states containing the active control points selected during phase one. At most one subprocessor is applied to a given process state during any phase two step. Although many process states in a given system may be requesting the same subprocessor, they each have it applied in parallel during phase two. Whether they are really serviced in sequence or in parallel is an implementation decision that affects only the duration of phase two since no inter-process interactions in this network system can occur until phase two has completed. All processes containing active control points will have sub-processors applied prior to the end of phase two.

Each sub-processor transforms only the network process state to which it is being applied. The applied sub-processor thus defines the successor network process state. If a system service is requested (a non-local transformation), the sub-processor will complete its network process step leaving a message, requesting the service, in the local interface buffer for subsequent phase three processing. Note that network processes in the same system will proceed in synchronous parallel with each other, each completing one process step in each system cycle. Network processes in different systems will run asynchronously parallel.

During phase three all requests for non-local services which were left in the interface buffer are acted upon. There are four types of these services, message transmission, resource transmission, process state transmission, and system modification. Phase three ends when all such services have been completed and the interface buffers are cleared.

If the interface buffer contains a request for message transmission, the associated message is removed from the buffer and "broadcast" to all systems but addressed to a particular control point buffer. The system containing the control point (destination) buffer will receive it; other systems will ignore the broadcast. Remember that a control point may have several destination buffers associated with it and that they are maintained as system state elements. Implementers, of course, are free to keep records of which system a control point resides in to permit them to transmit to only one system if they so desire. The relationship of these network system state elements is shown in Figure 2. The message has a network standard format and is represented as a string over a network standard character set.

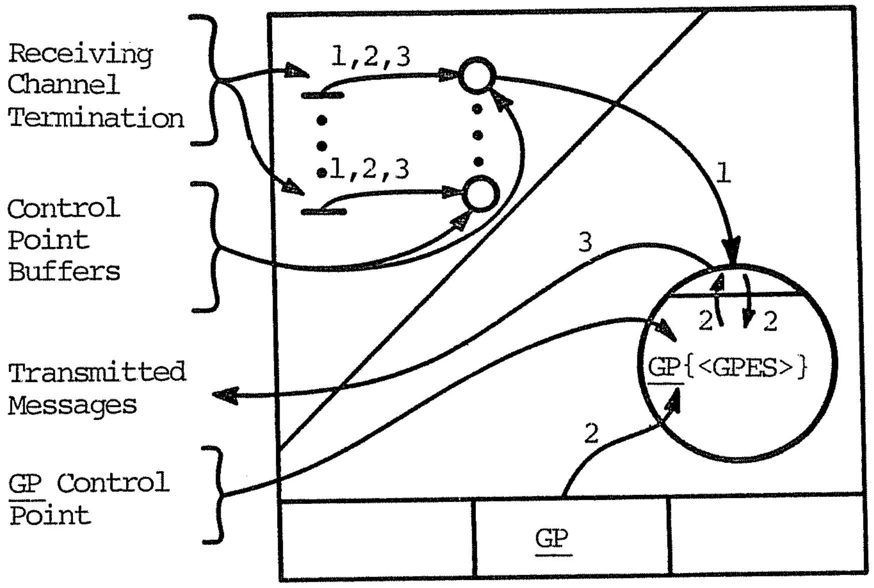


Figure 2: Intra-system message paths. The integers labeling communication transitions identify the system phase of that transition. GP is the sub-processor that interprets the network system language. <GPES> contains GP state information. The system step to transfer channel termination message to the corresponding control point buffers can be bypassed for messages arriving, at end of phase 3, if they are to be delivered on the next phase 1.

Although messages broadcast during phase three are subject to unspecified transmission delays (unless messages are intra-system), the order of transmission between any two systems is preserved in perception. In the receiving system, messages are placed in channel termination buffers and used to update control point buffers during every phase. A new message for a given buffer will overwrite the current contents (if any) of that buffer. If the destination control point is disarmed, the message will not be received at all. Figure 3 gives some examples of message interactions. In Figure 3 process A sends a message to process B in another system. B responds with a message for A. Process C sends a message to process D in the same system.

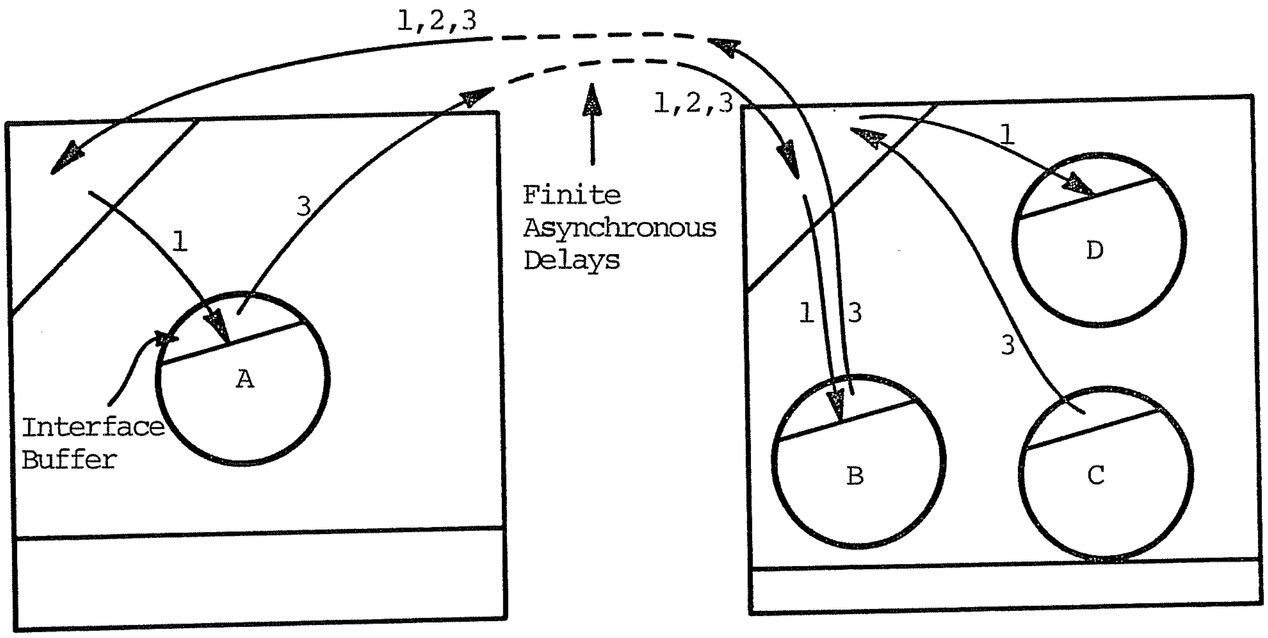


Figure 3: Interprocess Message Paths.
The Integer Labeling Communication Transitions.
Identify The System Phase of that Transition.
The Letters Identify Network Process States.

Note that the direct mechanism provided for interaction between processes involves message transmission from a network process state to some control point in a network process state. By controlling the access to destination control point names, network processes can be arbitrarily isolated so that uncooperative behavior effects can be localized. In addition, as far as network processes are concerned, message transmission is transparent to system boundaries and network processes may cooperatively move from system to system without losing communication or restricting their interactions.

5. Network Processes

Since non-local transformation services must be provided by the system processor rather than by subprocessors, the system processor designer must create some standard structures in each network process state which remain invariant to all sub-processor transformations. The interface buffer is just one example of such structures. Subsequent decisions by other designers must obey such constraints on network process state structures. Decisions concerning other process state structures are deferred to sub-processor designers.

A network defines the local environment and address space for sub-processor transformations. The interface buffer serves to factor the sub-processor transformations which are local to the process state from the system processor services which have effects outside of the process state. A process state will contain one or more control points, the status of which may be modified during system phase one as a result of message delivery or in system phase two as a result of the actions of an applied sub-processor. In addition to the role of control points in interprocess communication, control points also serve to delimit uniquely a portion of the network process state, called an <u>expression state</u>. Within this part of the process state the designers of the corresponding sub-processor are responsible for defining both the representations and the transformations which that sub-processor will carry out on those representations. All of the state information required for a sub-processor to continue a computation must be part of any corresponding expression state. A network process state thus contains an interface buffer, a set of expression states and one or more control points.

By constraining all virtual inter-process interactions to communications, and providing complete control over communications via restricted access to destination names, the network designers can supply sufficient mechanisms for application designers to isolate (to the desired extent) uncooperative computations. Having provided for worst case isolation, the network designers have an equal responsibility to provide for maximally cooperative computations. With the restriction that at most one subprocessor will be applied to a given process state in system phase two, we can permit multiple expression states of multiple types to interact as desired by their subprocessor designers and even access and transform structures and values in other expression states in the same process. We must insist that access and transformation of a given type expression state by subprocessors of different type obey the rules established by that given type.

The critical section conflicts that can arise in intra-process transformations by competing expression states are not (and cannot be) prevented by network designers who must instead provide sufficient primitive operators so that any desired cooperative conflict resolution technique can be used. The network designers do not have to deal with such intra-process conflict, or guarantee that the conflict will be resolved, since the process will still be well-defined and any side effects can be restricted to the process with the conflict. Multiple expression states in a network process state may be used as cooperatively as desired with minimal constraints from network designers.

Within an expression state we can distinguish between explicit "go to" executions and implicit "fault" conditions. Both forms change the "instruction counter" values of an expression state under rules specified by the corresponding sub-processor designer. Both forms have effects local to the containing expression state.

Similarly we can distinguish between explicit "move control point" executions and implicit "activate control point" (by internal command or external message receipt) conditions. Activations of multiple control points in a network process state represent a hierarchical interrupt system with the associated expression states serving as interrupt handlers. The movement of control points among expression states represent a scheduling operation such as for co-routines, tasks, Simula "processes", etc. Both of these forms are local to the process state.

We can also distinguish explicit and implicit inter-process movements of control points as resources (controlled access objects) as discussed in a later section on resources. An application program thus has a wide range of structures to use as benefits the application, assigning potentially uncooperative computations to isolatable network processes (running either synchronously or asynchronously) and exploiting the advantages of cooperative computations by putting them in a single process. A network process is capable of supporting all computations that can be carried out on a conventional single processor, multiprogramming system.

There is a sub-processor, called GP, in every network system. The GP expression state can be programmed, in the network system language, to provide or request all network services. GP expression states thus can specify invariant computations despite movement of the containing network process. The network system language thus plays many roles such as the following:

a) The network job control language.

b) The network high level "machine" language.

c) The network implementation language for operating "systems".

d) The base language for definitional extension via source language macros or compilers.

e) The base language for evolutionary augmentation.

f) The system invariant computation language.

A given programmer may use GP only as a definitionally extendible job control language, while suppling Fortran programs to a "type conversion" compiler that produces physical node dependent programs just as he is accustomed to do. The compiler in such a case would produce an "execute only" type value whose internal structure is virtually inaccessible. The execution efficiency could thus be unchanged. However a vastly more general computational environment structure can be specified for new applications. There is no unique system language required by the network system design and this design will work with many language structures. A description of a network system language, Aminol, is beyond the scope of this discussion. Indeed, Aminol will allow the definition and introduction of new sub-processors with their corresponding expression states so that the network system processor itself can become multilingual.

As a result of this design, the freedome to create a highly structured and locally managed process, while minimizing the operating system interference which is normally required to control side effects on other processes, can be delegated to application programmers. Although a process has almost total control over the management of its own operations, some other process must be able to control interactions outside a process, even without the cooperation of the process itself, in order to resolve conflicts.

The problem of deciding when a given interaction is improper can only be resolved by another process aware of the interactions. The system cannot resolve conflicting claims of one process with respect to the behavior of another process since it may require intimate knowledge of the specific applications. In order to guarantee resolution there must be a responsible authority who can control and manage interactions between the warring parties. Our system imposes a hierarchy of uniquely designated responsible authorities in the form of a network process tree, as in Figure 4, to make such an arbitration. All authority rests initially in the root of the tree. Although delegation to a child process is allowed, each root of a sub-tree remains responsible to its parent for the interactions of the processes in that sub-tree. An uncooperative root of a sub-tree will still be accountable to its parent process. One of the responsibilities of the network designer is to ensure that a cooperative process can restrict inter-process interactions of its sub-tree of processes.

The lines of responsibility in Figure 4 that define the process tree must have a basis in an ability of each process to control, and in worst case, isolate and kill, its sub-tree of processes. These control mechanisms are provided by management of "rights to..." as protected virtual resources using the facilities of the AO (accountable object) sub-processor as discussed below.

The process tree can both grow, by creating new "leaves," and shrink, by destroying "leaves." An existing process may move, as permitted by the parental authority, to any network system known to that parent. Thus both the process tree and its distribution over the network systems can change dynamically as a result of process computations. Since these are intrinsically non-local operations, the GP sub-processor can only request them. The responsibility for carrying them out has been delegated to a PR (Process Receive) process and its corresponding PR sub-processor. Each non-foreign network system will contain one of each. The GP requests for such service thus take the form of messages to the local (to the network system) PR process. Process birth and death are described in Figures 5 and 6.

Processes may be moved from system to system in order to control parallelism, to exploit specialized system implementations, to access special sub-processors existing in particular systems, or to carry out inter-process communications local to one system instead of by inter-system communications. Accesses to a special data base may be more efficient in a particular system. Such process state transmission is inherently an operation not local to one process or one system. When a process executes the appropriate transmit operator, the request is placed in the process interface buffer. The destination system is specified by referencing the name of a process contained in that system. Process names are protected objects managed as resources. During phase 3, the system processor transforms the process state into a "message" and places it in a buffer for that system's control point. The somewhat complex chores involved in transmitting and receiving process states are fulfilled cooperatively by the involved PR sub-pro-

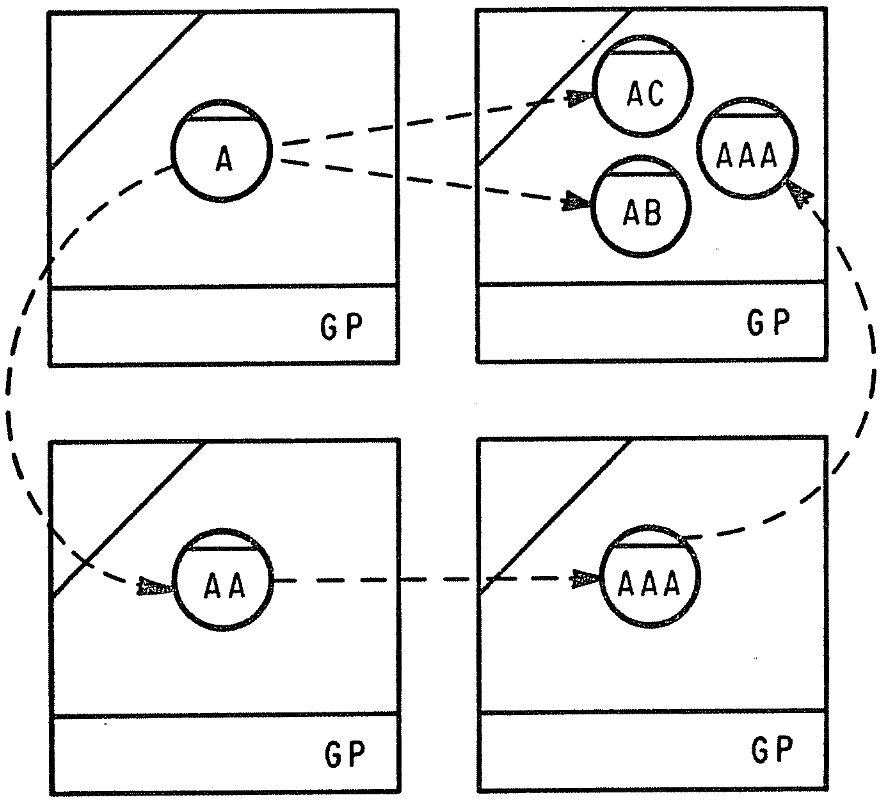


Figure 4: Network process tree lines of authority
Letters in process states identify the processes.
Conflict between process "AAA" and "AB" can only be resolved by their least common ancestor, process "A".
By definition, no conflict can arise between process "AA", "AAA", and "AAAA" since each ancestor can arbitrarily control its descendent.

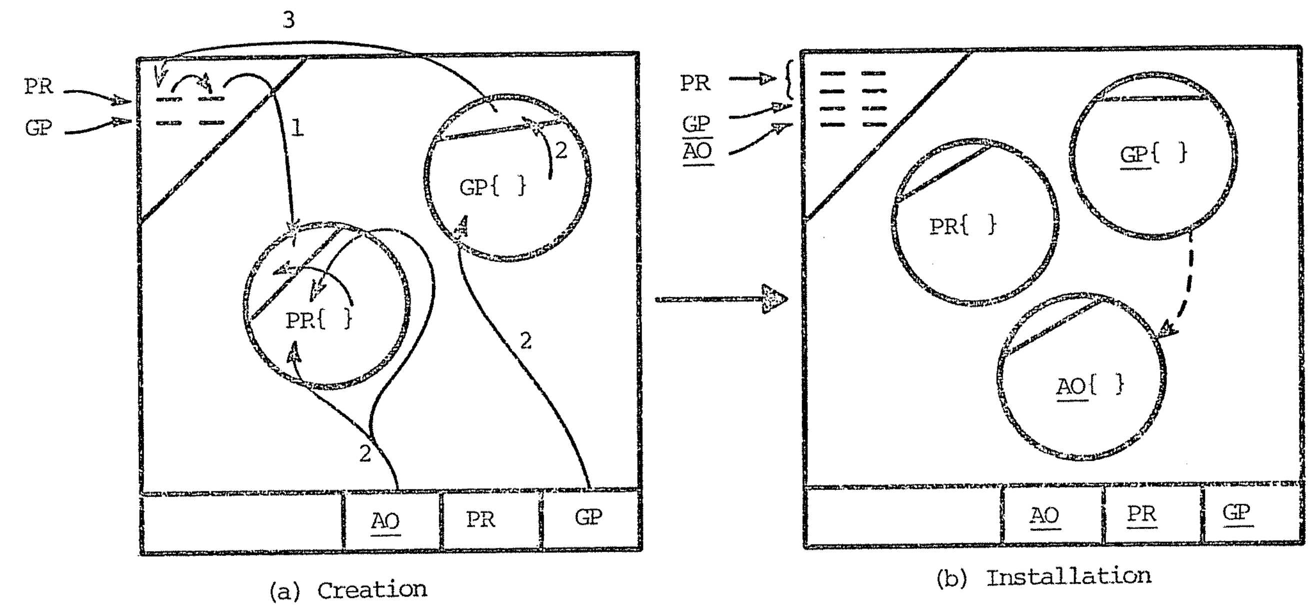


Figure 5: Network process birth.
The integers associated with arrows identify system phases. The transformation between (a) and (b) occurs in the last system phase 3. Dashed arrow represents process tree branch.

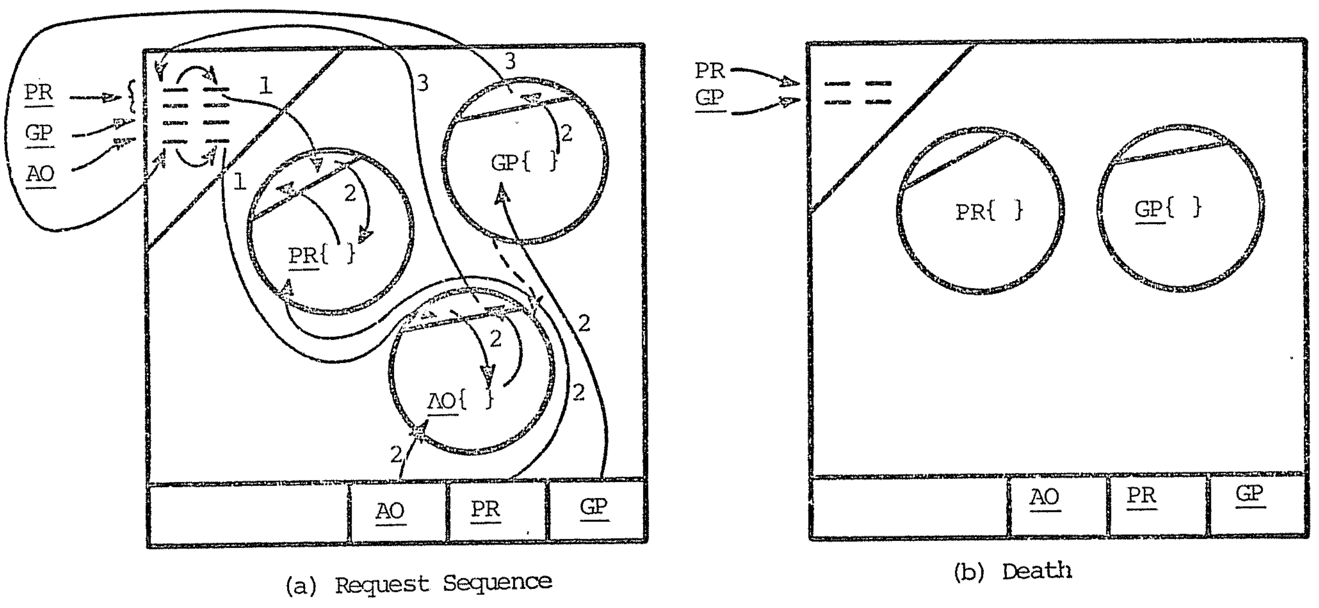


Figure 6: Network process death.
The integers associated with arrows identify system phases. The "leaf" process is deleted in the last system phase 3. Dashed arrow represents process tree branch.

cessors. A unique PR process state containing only a PR control point and PR expression state is included in the system processes of each non-foreign system in the network. Since we cannot guarantee the integrity of process states in foreign systems, we do not allow process states to be either transmitted to or received from foreign systems.

Communication of process states by the PR subprocessor must be reliable. We must permit an implementation to refuse to receive an additional process state without damaging the process in the transmitting system. This is done by not destroying the process state in the sending system until the receiving system acknowledges receipt of it. If a rejection response is received, the process state is "revived" in the source system and the original transmission operator in the revived process is faulted. Process management may then do as they like to ameliorate the problem.

The conversion of a process state to a message in the source system and the inverse operation in the destination system can easily be defined formally in the network. The corresponding operators in an implementation will of course be implementation dependent translators of the network defined form.

6. Resource Management

A process may cooperatively transmit any object to another process, but once the object is in the destination process state it is at the mercy of that process state. If the transmitting process wishes to restrict the access, transformation and disposal of the transmitted object we must provide facilities to guarantee the validity of the source-imposed constraints even if the destination process is uncooperative. All control of inter-process interactions is based on the distribution of "rights to..." by the root of a sub-tree. We will call all such constrained objects resources in the virtual systems, and delegate the responsibility for enforcing those constraints to the AO (accountable object) sub-processor. The AO designers in turn, delegate all possible responsibility to process management defined programs, while remaining within the constraints placed on them by the network designers. There are three services that must remain with the AO sub-processor: that of reliable (guaranteed cooperative) communication, that of protecting access controls of a resource, and that of pre-emptive return of resources from uncooperative sub-trees. In order to ensure resource constraints, each network process state will contain a unique AO control point and AO expression state. Thus resources are passed to guaranteed cooperative AO expression states that enforce the constraints whether the process is cooperative or not. Thus the process tree "lines of responsibility" are embodied in cooperative AO communications and all resources are constrained to movements over the process tree. We thus prevent any conflicts of authority over a resource since, for each sub-tree, the root process is uniquely responsible. The detailed derivation and design of the AO sub-processor and its associated expression state is given in (3). We will only indicate the nature of the design here.

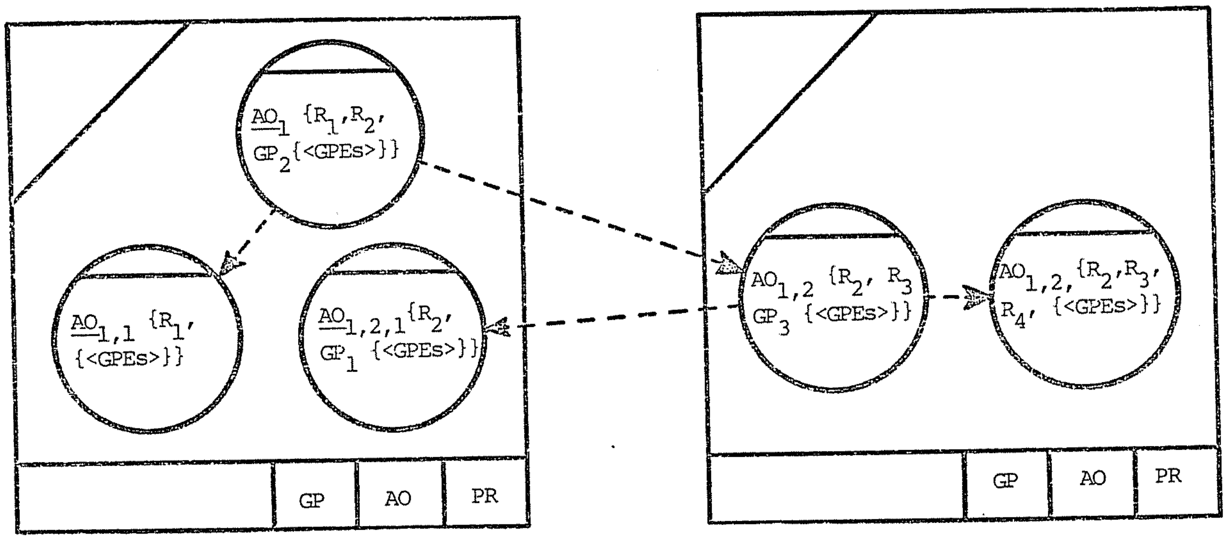
Unblockable communications are guaranteed by assigning the AO control point in each process state the highest priority in the process. Guaranteed asynchronous communication is provided by creating AO receiver buffers for the parent and for each of the children. In order to acknowledge message receipt, a set (at least one) of response buffers are provided in the transmitting process. The number of processes transmitting an acknowledgement to a particular process is restricted by requiring that any such transmission uniquely designate a particular response buffer. Using these features it is possible to define a communication protocol that will provide the required communication services for resource management.

The AO expression state contains a set of resources and each resource contains an object and an ordered set of access procedures. All access to the object by programs of the containing process must be made by the execution of the ordered set of procedures. A process may create a resource by specifying any object and any access procedure. AO conventions guaranteed the protection of the resource without constraining process management's resource specification.

The access procedure may allow the allocation of the resource object (or a part of it) to a child process, while adding a new access procedure which must be executed to gain access to the previously specified access procedure. Thus additional, properly nested, access constraints may be added during alocation to a child process. Figure 7 presents an example of the resource relationship.

Process management is delegated responsibility for the normal management, allocation, transformation and return of resources. The GP sub-processor will include suitable primitive operators for this purpose. When cooperation breaks down, pre-emptive return via AO services is required. Since a resource may be fragmented and sub-allocated in pieces according to the rules and access mechanisms defined by the resource creator, AO only returns the pieces and lets the owner put them back together.

The transmission of unwanted or spurious messages is an explicit non-local process interaction which must be controllable. Any transmitting operator is required to have, as an operand, a resource containing a "right to transmit" to a particular control point buffer.



AO1 {R1,R2, GP2{<GPEs>}}
AO1,1 {R1, {<GPEs>}}
AO1,2,1 {R2, GP1 {<GPEs>}}
AO1,2 {R2, R3 GP3 {<GPEs>}}
AO1,2, {R2,R3, R4, {<GPEs>}}
GP
AO
PR
GP
AO
PR

Figure 7: Interprocess Resource Relationships

The creation and allocation of such permits is under the control of process management using GP programs. This permits a parent to deny to a child the ability to interact explicitly with any other process except that parent or a sub-tree created by that child.

The tree structure is defined by the names (subscripts) of the AO control points. The AO expression state contains the set of resources and a GP expression state. A prime on a resource indicates an accessing program which has been added by other than the resource creator. Since each accessing program is additive, restrictions may be added but never removed by a child.

Implicit interaction between processes may occur as a result of conflicts for a particular set of implementation resources (physical). These interactions occur as the result of the exhaustion of some bounded physical resourced used in the representation of process states and must also be controllable by process management. Although we must not interfere with implementation management, we are entitled to delimit the scope of competition for such resources and define responsibility for its exhausion. By giving a child explicit delegation of a "limited competition permit" resource, a parent may restrict the competition of any child with that parent for such essential resources. When the limit of such a physical resource is reached, the effects are constrained to the sub-tree that was most locally restricted and will appear to the process as an "inaccessible object" in a process state. Inaccessible objects play a role similar to end of file marks on a tape drive and their detection by process management allows explicit programming for amelioration of the problem of recovery if desired. Competition restriction allows a parent freedom from a child's excesses in use of representational resources.

An example of the use of this restricted competition mechanism could be the partial control, in demand paging implementations, of replacement page selection.

7. Operating Systems

Network design responsibilities have been factored between the implementation and process managers. This requires a clean factoring of network resources (e.g., a file) and implementation resources (e.g., a drum) upon which the logical network resources are maintained. Implementation designers clearly must be delegated responsibility for mapping network resources onto implementation resources. It is thus necessary to have two different operating systems, one for the virtual systems, and another for each of the physical node systems. The current size, complexity, and unmanageability of contemporary operating systems is in large part due to trying to meet these diverse goals with one operating system.

Physical operating systems may vary widely from node to node and may be developed independently subject only to minimum constraints placed on them by the network designers. Designers of each physical node can select some utilization function and then manage the physical resources in a way that optimizes this function. The implementation programming language used should provide easy exploitation of machine dependent resources and provide a highly optimized machine dependent implementation. In addition, at each node programmers may be under one management and can produce cooperatively structured implementations by informal, but very hard to enforce, conventions. As long as the implementation can be debugged prior to its productive use in the network, inadvertent failures of cooperation can be prevented. Highly structured hierarchical layers of queue networks could be constructed as cooperative sequential processes and deadlock problems which can not be cooperatively prevented, can be cured by sacrificing jobs that can be re-run.

Like the implementers, the network managers (users) may have goals which very markedly from process to process. The factorization above makes it possible for each of them to pursue these goals subject only to minimal constraints placed on them by network designers and network implementers. There is usually some utilization or reliability function which they are trying to optimize during their management of user created resources. They need a programming language that provides for the easy exploitation of machine independent resources and produces highly optimized (with respect to application measures), but machine-independent, network processes as distinct from the implementers who are dealing with machine dependent resources and implementations. In the design presented here each process is subject to constraints imposed on it by its parent process. It can be held responsible for its actions while being free to manage its own affairs. It is possible to protect other network processes from interactions they do not wish to experience. Many users need to be able to develop programs simultaneously even though they are not all under the same management, and debugging operations must be able to proceed in the network while it is up and performing services for other processes. Inadvertent errors, as well as malicious non-cooperation by one process must not jeopardize system performance for others. When user program structures are based on cooperative hierarchies they often either become unpoliceable or they collapse under the required policing constraints. This independence means that deadlock problems cannot be prevented or cured by reasonable algorithms imposed by the system. What is required instead are tools and facilities that allow delegation of such decisions to process managers who can choose appropriate tactics. In order to allow such application dependence network and implementation resources must be divorced. Although there still exists a "master/slave" mode, each process can now become the local operating system for itself and its children, subject only to the authority and constraints of its parent. This is advantageous since no global, premature, decisions need be made by a parent because now they can give their children the freedom to optimize their own operations while still retaining necessary control over them in case they go astray.

In order for process managers to meet their commitments they need a high level resource creation and manipulation language. In our network each system will provide a common subprocessor (GP) to interpret this virtual system language. Conventional and often often unmanageable job control languages, with their many options and lack of general freedoms provide too rudimentary and specialized control of resources for processes engaged in defining operating environments for their children. A high level system language for network operating system implementation is required for this purpose.

Other application languages may be provided either by compilers producing output for the GP subprocessor or for other specialized subprocessors. To provide for such compilers, the system language interpreted by GP must, of course, be extensible. It must also be augmentable by implementation adaptions (possibly via micro-programming) if specific efficiency requirements for specific processes, as well as general evolutionary processes, are to be permitted.

The design and specification of such a system language has been substantially completed and will be reported on subsequently. A prototype implementation of one complete network system will be completed in the near future. It should be pointed out that the overall network structure described here is not dependent on a specific GP design for its validity.

8. User Freedom

In the previous sections we briefly introduced some of the features of the network design. In this section we would like to present some of the freedom which these features provide the user. As can be seen from the design features presented, the network designers have not constrained in any way the design of user algorithms. The network is abstracted, both from any particular application of it and from any particular implementation of it. Different physical nodes supporting systems in the network, may use different physical assets to do so, with no concern for the users other than how it affects the costs of running their processes. The mechanisms invoked by a user to cause a subprocessor to be applied to a control point/expression state pair, or to use interprocess communication services are standard in each non-foreign networks system. As a result, process states may be moved between systems using the services of the PR process and still be able to run in other systems without explicit changes to reflect the new system's implementation conventions.

Another important freedom provided by this design is the ability of a user to construct a single process environment containing multiple expression states, of different specialized subprocessor types, which may interact cooperatively. Very few constraints are placed on the subprocessors applied to such cooperating expression states by the network design other than the prevention of true race conditions (only one subprocessor at a time is applied to a process state) and the prevention of a violation of resource integrity (because of AO design). Adaptation of user specialized cooperative intra-process interactions is thus minimally constrained.

Control point communication provides several advantages to the user. First, since control points are paired with expression states, the destination of an interprocess communication can be bound to a particular expression state in a particular process. A sender need not be aware that a destination control point has been moved to a new process or that a whole process, containing the destination control point has been moved to another system. For the implementer, control point communication permits the maximum freedom to use any method of intersystem communication desired. The network could easily be implemented on the ARPANET (2, 4, 7, 9, 10) or on most of the other networks descirbed in (1) and (5). Since the network design does not tell the implementer how to handle messages. The interrupt capability provides a significant advantage in terms of interprocess interactions. A control point/expression state can stop processing and go to sleep knowing that lower priority control points in that process will proceed until the higher priority control point either receives a message or is designated to run by a subprocessor operating on one of the other expression states in that process.

An advantage of using the system/subprocessor and process/control point/expression state concepts is that it becomes almost trivial to introduce new subprocessors into the network. The old processes still run the same while new processes can take advantage of the additions. The scope of the effects of such introductions will be limited to the processes introducing such subprocessors or to processes allocated the right to use them. Another advantage of providing a common interface is that it permits a user to move his process to other systems.

In addition to (8), the network design is more completely covered in (3) and other papers in preparation. It is of course not possible here to describe all of the aspects of this project. The development of this design was done with a cleanly factored set of design constraints which were abstracted without any particular technology or application in mind. This is one of the more important reasons why the design is a solution to many problems of manageability of processes in a network, of application generality, of process portability between systems in the network, and of network survival between changes in technology.

The design is fully implementable with the worst implementation being a complete simulation. An implementation currently exists of a single system, multiple process, and multiple expression state with multiple control points. The simulation of the functions of inter-process communication and control performed by the system processor is a relatively minor problem since the system processor implemented supports all but inter-system messages. It is not these functions which create implementation complexities. With the advent of microprogramming, overhead can be reduced, particularly in the implementation of subprocessors. As hardware, including firmware, becomes less expensive, the system processor functions of communications and subprocessor application can be easily implemented in them. Such implementations of these functions will be able to perform them in the range of several gate times if desired. The advantage of hard-wiring all subprocessors is something which should be further investigated. The choice between microprogrammed or hardwired subprocessors becomes one of balancing efficiency against flexibility.

Another important aspect of this design, not emphasized here, is that the network as designed has also been formally defined. It is important that the more formal properties of any design be investigated. A formal system provides us with an unambiguous, machine independent way of defining our results. Without such a system, the design might not be understandable by either the implementer or the user. Machine independent definitions are particularly necessary since we are interested in networks of systems and asynchronous computations. As a result of the formal definition system used, we get a well-defined concept of process and process step. The use of an interpreter for the definition system permits the design to be debugged. Analysis of the system definitions provides valuable insights into the process behaviors supported by the network.

One area introduced in (8) and to be pursued in subsequent reports is the possibility of permitting user control over certain modifications to the network design. This would include modifications of the formal definition to include new systems, new subprocessors, or even new operators in current subprocessors. One of the important constraints on such modifications is the verification that they will not improperly affect the processes in the network for which the user performing a modification is not responsible.

This brief introduction into the network design only touches on the basic structure of the network. The references can provide the interested reader with a detailed analysis of the design. We are interested in providing a network of interacting digital computer systems and structures to support general processes. This design provides user control over potentially uncooperative processes in a multiprocess computation. This includes the sharing of capabilities and permitting interactions of processes resident in different virtual systems. The design presented here has, we feel, accomplished these goals.

References

1. Bell, C. Gordan "More Power by Networking," IEEE Spectrum, Feb. 1974, pp 40-45.

2. Carr, C. Stephen, Crocker, Stephen D. and Cerf, Vinton G. "HOST-HOST Communication Protocol in the ARPA Network," Proc. AFIPS SJCC, 1970, Vol. 36, pp 589-597.

3. Cowan, George Jr. Management of Resources in a Potentially Hostile Environment (Logical and Physical). Ph.D. Thesis, University of Wisconsin, Madison.

4. Crocker, Steven D., Heafner, John F., Metcalfe, Robert M., and Postel, Jonathan B. "Function-Oriented Protocols for the ARPA Computer Network," Proc. AFIPS SJCC, 1972, Vol. 40, pp 271-279.

5. Farber, David J. "Networks: An Introduction," Datamation, April 1972, pp 36-39.

6. Fitzwater, D. R. and Hintz, C. A. A System for the Formal Definition of Digital Systems. CS Tech. Report #141, The University of Wisconsin Computer Sciences Department, 1971.

7. Heart, F. E., Kahn, R. E., Ornstein, S. M., Crowther, W. R. and Walden, D. C. "The Interface Message Processor for the ARPA Computer Network," Proc. AFIPS SJCC, 1970, Vol. 36, pp 551-567.

8. Kramer, John F. A General Structure for Uncooperative Processes Distributed Over a System Network. Ph.D. Thesis, University of Wisconsin, Madison, 1973.

9. Ornstein, S. M., Heart, F. E., Crowther, W. R., Rising, H. K., Russell, S. B. and Michel, A. "The Terminal IMP for the ARPA Computer Network," Proc. AFIPS SJCC, 1972, Vol. 40, pp 243-254.

10. Roberts, Lawrence G. and Wessler, Barry O. "Computer Network Development to Achieve Resource Sharing," Proc. AFIPS SJCC, 1970, pp 543-549.

APPENDIX D

D.1 Introduction

The following is an example of a functional specification using the notation developed in Section 3. The system specified is the network described in Section 5, and detailed in Appendix C. It should be noted that this system was originally specified using a different formal specification technique, thus the resulting design may not allow for the cleanest re-specification using the technique of Section 3.

Two specifications of the system will be presented here. The first is a high level specification whose functions are defined in terms of very high level primitives. The second specification is a more detailed version of the first. The form of presentation of both specifications is the same. First, the functions are defined with a short description accompanying each definition. Following this, the function definition tree and process graphs are illustrated. And finally the functions of the specification are summarized in a table. Preceding both specifications (and should be considered part of each) is a definition table of the value spaces and component spaces used in the specification.

D.2 The Specifications

Both specifications consist of two state successor functions, 'SYS' and 'REAL WORLD'. SYS specifies a single network system and REALWORLD specifies the rest of the network as it appears to a single system.

Generally speaking, an application of SYS does the following:

(1) Makes subprocessor and message buffer selection for each process state in the system.

(2) Applies the subprocessors, leaving resulting messages in interface buffers.

(3) Transmits messages.

The REALWORLD system contains a transmitter and a receiver and a packet of messages which have yet to be delivered for each system in the network. Roughly it operates as follows:

(1) All receivers are fired in parallel picking up a packet (if any) sent from each system.

(2) These packets are merged into one packet with an arbitrary choice made between messages for the same destination.

(3) The resulting packet is distributed amongst the systems updating a packet containing all of the messages for each of the systems which have yet to be delivered, new messages for such a packet over-write old messages with the same destination.

(4) The updated packets are all transmitted in parallel from the REALWORLD system and any which do not get delivered form the undelivered packet for that system.

The reader is urged to refer to the function trees and process graphs and also to the description of network system interaction in Appendix C.4 when reading the specifications.

D.2.1 Definition of Value Spaces and Component Spaces

VALUE SPACES

V_p	$\equiv$ Space of process states
V_{mB}	$\equiv$ Space of message buffers
V_{IB}	$\equiv$ Space of interface buffers
$V_{messages}$	$\equiv$ Space of messages
$\{\$\}$	$\equiv$ Space containing $
$\{\varepsilon\}$	$\equiv$ Space containing ε
$V_{\overline{p}}$	$\equiv V_p \cup \{\$\}$
$V_{\overline{mB}}$	$\equiv V_{mB} \cup \{\$\} \cup \{\varepsilon\}$
V_{TO_i}	$\equiv$ Values of messages directed to system i
V_{FROM_i}	$\equiv$ Values of messages sent from system i
V_{NEW}	$\equiv$ Values of messages in newly arrived packet (merged from all systems)
N	$\equiv$ Number of systems in network

$i \leq N$.

$t \equiv$ TRUE, $f \equiv$ FALSE

Σ_p	$\equiv$ Component space of V_p
Σ_{mB}	$\equiv$ Component space of V_{mB}
Σ_{IB}	$\equiv$ Component space of V_{IB}
$\Sigma_{message}$	$\equiv$ Component space of $V_{message}$

$\Sigma_{\overline{p}} \equiv$ Component space of $V_{\overline{p}}$

$\Sigma_{\overline{mB}} \equiv$ Component space of $V_{\overline{mB}}$

$\Sigma_{SYS} \equiv \Sigma_p \times \Sigma_{mB}$

$\Sigma_{TO_i} \equiv$ Component space of V_{TO_i}

$\Sigma_{FROM_i} \equiv$ Component space of V_{FROM_i}

$\Sigma_{NEW} \equiv$ Component space of V_{NEW}

$$\Sigma_{REAL} \equiv \prod_{i=1}^{N} \Sigma_{TO_i}$$

(Note: $\prod_{i=1}^{N} \Sigma_i \equiv \Sigma_1 \times \Sigma_2 \times \dots \times \Sigma_n$)

D.2.2 High Level Specification

The function tree and process graph for SYS appear in Figures 1 and 2 for the REALWORLD in Figures 3 and 4.

D.2.2.1 High Level Specification of SYS

(1) SYS: $\Sigma_{SYS} \rightarrow \Sigma_{SYS}$

$SYS(\sigma_{SYS}) \equiv SYS_1(\sigma_p, \sigma_{mB})$

Comment: The state-successor function SYS is composed of a single component function.

(2) $SYS_1: \Sigma_p \times \Sigma_{mB} \equiv \Sigma_p \times \Sigma_{mB}$

$SYS_1(\sigma_p, \sigma_{mB}) \equiv$ Phase 3 $(SP_1(STATUS(\sigma_p, \sigma_{mB})))$

Comment: SYS_1 is defined in terms of three component successor functions PHASE 3, SP_1 and STATUS.

(3) STATUS: $\Sigma_p \times \Sigma_{mB} \rightarrow \Sigma_{\overline{p}} \times \Sigma_{\overline{mB}}$

STATUS$(\sigma_p, \sigma_{mB}) \equiv$ PRIMITIVE

Associated with each process state is at most one message buffer, which one it is is determined by STATUS. The STATUS primitive will return pairs of (process state, message buffer) such that the process state is requesting that message buffer and the buffer is a buffer for the highest priority control point in the process state. If a message buffer is not requested a dummy is returned: ($, message buffer). If a process is active and has (or wishes) no messages: (process state, ε) is returned, and if a process state is not to have a subprocessor applied it returns (process state, $). Thus STATUS does the selection of the appropriate control point expression state in a given user state for subprocessor allocation, and associates the appropriate message with the process state. It returns a dummy pair for unused messages and also for unused process states.

(4) SP_1: $\Sigma_{\overline{p}} \times \Sigma_{\overline{mB}} \rightarrow \Sigma_p \times \Sigma_{mB} \times \Sigma_{IB}$

$SP_1(\sigma_{\overline{p}}, \sigma_{\overline{mB}}) = SP_{11}(V_{\overline{p}}, V_{\overline{mB}})$

(5) SP_{11}: $V_{\overline{p}} \times V_{\overline{mB}} \rightarrow V_p \times V_{mB} \times \Sigma_{IB}$

$SP_{11}(V_{\overline{p}}, V_{\overline{mB}}) \equiv$ PRIMITIVE

SP_{11} applies the appropriate subprocessors to the elements of $(\sigma_{\overline{p}}, \sigma_{\overline{mB}})$, delivering the messages (if any) and leaving messages in the interface buffer. On dummy pairs SP_{11} simply carries forward the process state in (process state, \$) pairs and the message buffer in (\$, message buffer) pairs. The value function models the parallelism of subprocessor allocation.

(6) Phase 3: $\Sigma_p \times \Sigma_{mB} \times \Sigma_{IB} \rightarrow \Sigma_p \times \Sigma_{mB}$

Phase $3(\sigma_p, \sigma_{mB}, \sigma_{IB}) \equiv$ PRIMITIVE

Phase 3 finishes up the process step by:

(1) Transmitting the messages to external systems

(2) Receive messages from external systems

(3) Merge received messages with old message buffers

(4) Merge message buffers with intra-system messages.

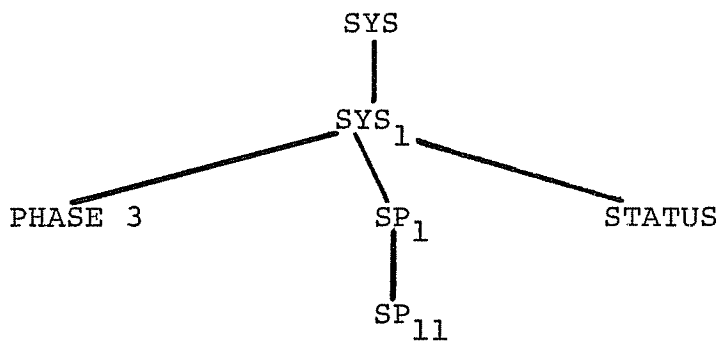


FIG. 1: Function tree for high level specification of SYS

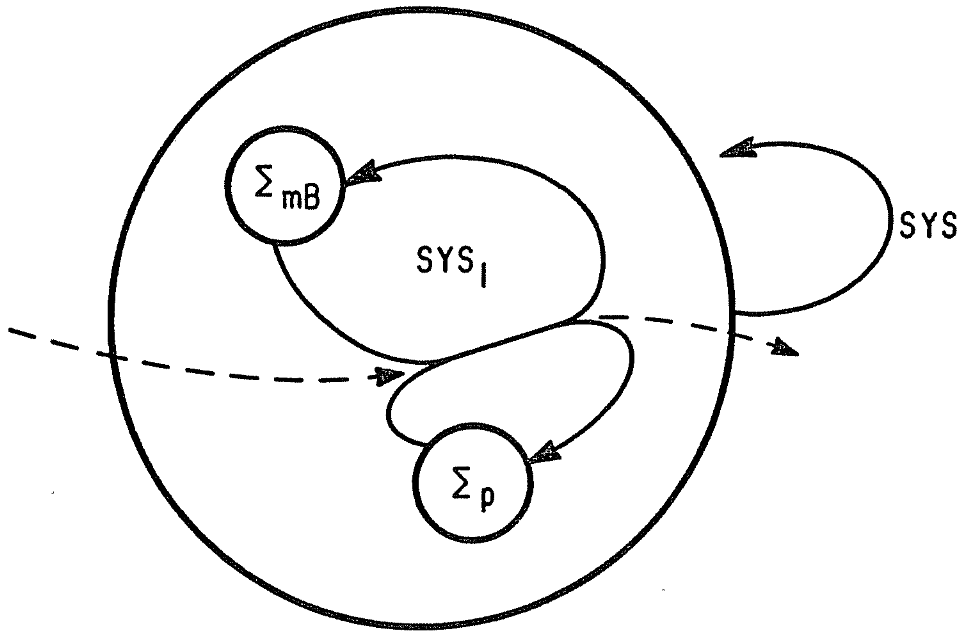


FIG. 2: Process graph for SYS

D.2.2.2 High Level Specification of Realworld

(1) Realworld: $\Sigma_{REAL} \rightarrow \Sigma_{REAL}$

Realworld(σ_{REAL}) $\equiv$ Realworld$_1$($\sigma_{TO_1}, \ldots, \sigma_{TO_N}$)

(2) Realworld$_1$: $\prod_{i=1}^{N} \Sigma_{TO_i} \rightarrow \prod_{i=1}^{N} \Sigma_{TO_i}$

Realworld$_1$($\sigma_{TO_1}, \ldots, \sigma_{TO_N}$) $\equiv$ Join (Packetin( ),
Trysend($\sigma_{TO_1}, \ldots, \sigma_{TO_N}$))

Comment: Realworld$_1$ merges the existing packets for each system with the newly arrived packet.

(3) Join: $\Sigma_{NEW} \times \prod_{i=1}^{N} \Sigma_{TO_i} \rightarrow \prod_{i=1}^{N} \Sigma_{TO_i}$

$\text{Join}(\sigma_{NEW}, \sigma_{TO_1}, \ldots, \sigma_{TO_N}) \equiv$ PRIMITIVE

Comment: Join updates each σ_{TO_i} with the messages in σ_{NEW} for the ith system, overwriting old messages for the same destination.

(4) Packetin: $\phi \rightarrow \Sigma_{NEW}$

Packetin( ) $\equiv$ PRIMITIVE

Comment: Packetin receives a packet from each system (if one was sent) and coalesces all received packets into one packet, choosing at most one message for each destination.

(5) TRYSEND: $\prod_{i=1}^{N} \Sigma_{TO_i} \rightarrow \prod_{i=1}^{N} \Sigma_{TO_i}$

$\text{TRYSEND}(\sigma_{TO_1}, \ldots, \sigma_{TO_N}) \equiv$ PRIMITIVE

Comment: TRYSEND tries to send the accumulative packet for each system to that system. All packets are sent in parallel.

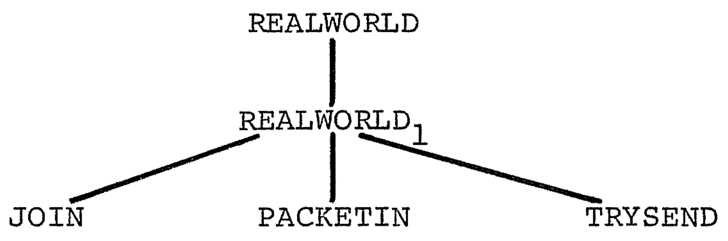


FIG. 3: Function tree for high level specification of REALWORLD.

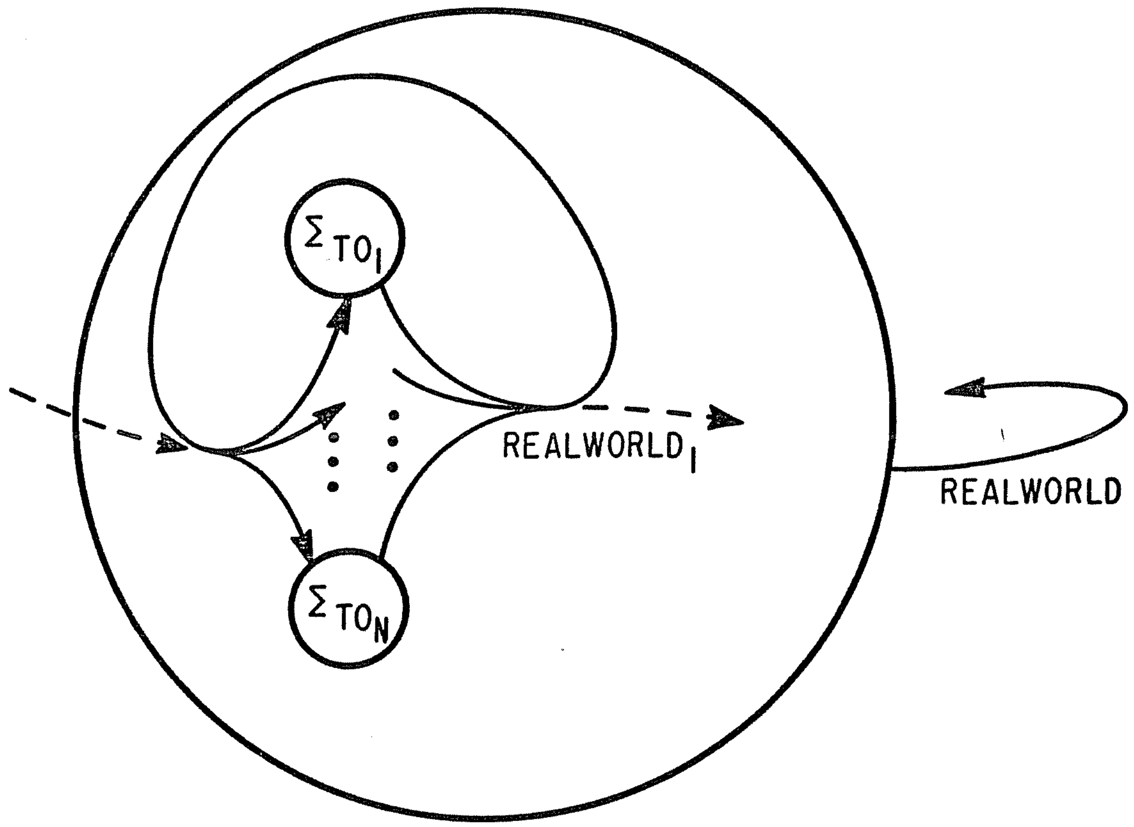


FIG. 4: Process graph for REALWORLD.

D.2.2.3 Function Table for High Level Specification

TYPE	MAPPING	DEFINITION
State Successor	SYS: $\Sigma_{SYS} \rightarrow \Sigma_{SYS}$	$SYS(\sigma_{SYS}) \equiv SYS_1(\sigma_p, \sigma_{mB})$
Component	SYS_1: $\Sigma_p \times \Sigma_{mB} \rightarrow \Sigma_p \times \Sigma_{mB}$	$SYS_1(\sigma_p, \sigma_{mB}) \equiv \text{Phase } 3(SP_1(STATUS(\sigma_p, \sigma_{mB})))$
Component	STATUS : $\Sigma_p \times \Sigma_{mB} \rightarrow \Sigma_{\overline{p}} \times \Sigma_{\overline{mB}}$	PRIMITIVE
Component	SP_1: $\Sigma_{\overline{p}} \times \Sigma_{\overline{mB}} \rightarrow \Sigma_p \times \Sigma_{mB} \times \Sigma_{IB}$	$SP_1(\sigma_{\overline{p}}, \sigma_{\overline{mB}}) \equiv SP_{11}(V_{\overline{p}}, V_{\overline{mB}})$
Value	SP_{11}: $V_{\overline{p}} \times V_{\overline{mB}} \rightarrow V_p \times V_{mB} \times V_{IB}$	PRIMITIVE
Component	Phase 3: $\Sigma_p \times \Sigma_{mB} \times \Sigma_{IB} \rightarrow \Sigma_p \times \Sigma_{mB}$	PRIMITIVE
State Successor	Realworld: $\Sigma_{REAL} \rightarrow \Sigma_{REAL}$	$Realworld(\sigma_{REAL}) \equiv Realworld_1(\sigma_{TO_1}, \ldots, \sigma_{TO_N})$
Component	$Realworld_1$: $\prod_{i=1}^{N} \Sigma_{TO_i} \rightarrow \prod_{i=1}^{N} \Sigma_{TO_i}$	$Realworld_1(\sigma_{TO_1}, \ldots, \sigma_{TO_N}) \equiv JOIN(PACKETIN(\;),$ $TRYSEND(\sigma_{TO}, \ldots, \sigma_{TO_N}))$
Component	JOIN: $\Sigma_{NEW} \times \prod_{i=1}^{N} \Sigma_{TO_i} \rightarrow \prod_{i=1}^{N} \Sigma_{TO_i}$	PRIMITIVE
Component	PACKETIN: $\phi \rightarrow \Sigma_{NEW}$	PRIMITIVE
Component	TRYSEND: $\prod_{i=1}^{N} \Sigma_{TO_i} \rightarrow \prod_{i=1}^{N} \Sigma_{TO_i}$	PRIMITIVE

D.2.3 Detailed Specification

Now we want to look at the system in some more detail, i.e., we decompose some of the functions that were regarded to be primitive in the high level specifications. The new function trees for SYS and REALWORLD appear in Figures 5 and 7. A process graph for the detailed specification of the component successor function SP_1 is shown in Figure 6.

D.2.3.1 Detailed Specification of SYS

There are no changes in the higher level functions, for sake of completeness of this detailed specification they are repeated here.

(1) SYS: $\Sigma_{SYS} \rightarrow \Sigma_{SYS}$

$SYS(\sigma_{SYS}) \equiv SYS_1(\sigma_p, \sigma_{mB})$

(2) SYS_1: $\Sigma_p \times \Sigma_{mB} \rightarrow \Sigma_p \times \Sigma_{mB}$

$SYS_1(\sigma_p, \sigma_{mB}) \equiv PHASE\ 3(SP_1(STATUS(\sigma_p, \sigma_{mB})))$

(3) STATUS: $\Sigma_p \times \Sigma_{mB} \rightarrow \Sigma_{\overline{p}} \times \Sigma_{\overline{mB}}$

$STATUS\ (\sigma_p, \sigma_{mB}) \equiv PRIMITIVE$

Comment: The function of STATUS is to pair all process states with their appropriate message buffer. There exists at most one for each process state. There are four possibilities:

(1) State v_p is requesting v_{mB} and v_{mB} is a buffer for the highest priority control point in v_p.

(2) State v_p is active and not requesting any messages, therefore STATUS returns (v_p, ε).

(3) State v_p is neither active nor has a message pending, thus STATUS returns $(v_p, \$)$.

(4) Message v_{mB} is requested by no v_p, STATUS returns $(\$, v_{mB})$

(4) SP_1: $\Sigma_{\overline{p}} \times \Sigma_{\overline{mB}} \rightarrow \Sigma_p \times \Sigma_{mB} \times \Sigma_{IB}$

Comment: In effect SP_1 will apply the correct subprocessor (if any) to the process states and messages (if any) resulting in new process states, message buffers and interface buffer values. To model the parallelism of what is effectively Phase 2 of the system step we will specify SP_1 in terms of four value functions. The SP_{1j} $(1 \leq j \leq 4)$ are case distinctions, i.e., their D_{1j} are disjoint and define a partition of the domain $(\sigma_{\overline{p}}, \sigma_{\overline{mB}})$; therefore we can use subtree selection (as formally defined in Appendix A). We define now:

$$SP_1(\sigma_{\overline{p}}, \sigma_{\overline{mB}}) \equiv (\sigma'_p, \sigma'_{mB}, \sigma'_{IB})$$

where

$$\sigma'_K = \bigcup_{1 \leq j \leq 4} \Big(\bigcup_{(x,y) \varepsilon (\sigma_{\overline{p}}, \sigma_{\overline{mB}})} P_k(SP'_1(x,y))$$

and $k \ \varepsilon \ \{p, mB, IB\}$

and

$$SP'_1 = ((x \varepsilon V_p \wedge y \varepsilon V_{mB}): \ SP_{11},$$
$$(x \varepsilon V_p \wedge y = \varepsilon): \ SP_{12},$$
$$(x = \$ \wedge y \varepsilon V_{mB}): \ SP_{13}, \ SP_{14})$$

Now let us discuss the four cases:

(5a) $SP_{11}: \ V_p \times V_{mB} \rightarrow V_p \times V_{IB}$

$SP_{11}(v_p, v_{mB}) \equiv$ PRIMITIVE,

Comment: SP_{11} applies Phase 2 onto a process state/memory buffer-pair in STATUS 3.1: it delivers message v_{mB}, and applies appropriate subprocessor resulting in a new state and possibly a message in the interface buffer.

(5b) $SP_{12}: \ V_p \times \{\varepsilon\} \rightarrow V_p \times V_{IB}$

$SP_{12}(v_p, \varepsilon) \equiv$ PRIMITIVE,

Comment: SP_{12} applies Phase 2 onto a process without associated message buffer (STATUS 3.2) in the same way as SP_{11} except no message is delivered.

(5c) SP_{13}: $\{\$\} \times V_{mB} \rightarrow V_{mB}$

$SP_{13}(\$, v_{mB}) \equiv v_{mB}$, carries forward unused messages

(5d) SP_{14}: $V_p \times \{\$\} \rightarrow V_p$

$SP_{14}(v_p, \$) \equiv v_p$, carries forward unchanged process states.

Comment: See process graph for SP_1, Figure 3.

The subprocessors have been applied so it remains to specify the message transmission and receival.

(6) Phase 3: $\Sigma_p \times \Sigma_{mB} \times \Sigma_{IB} \rightarrow \Sigma_p \times \Sigma_{mB}$

Phase $3(\sigma_p, \sigma_{mB}, \sigma_{IB}) \equiv (\sigma_p, \text{UPDATE}(\text{NEWIN}(\sigma_{mB}), \text{NEWOUT}(\sigma_{IB})))$

Comment: Phase 3 merely carries forward the process states σ_p, and updates the message buffers.

(7) UPDATE: $\Sigma_{mB} \times \Sigma_{messages} \rightarrow \Sigma_{mB}$

$\text{UPDATE}(\sigma_{mB}, \sigma_{messages}) \equiv \text{PRIMITIVE}$

Comment: Update merges the messages in σ_{mB} and $\sigma_{messages}$ with the provision that if they each have a message for the same destination only the message in $\sigma_{messages}$ is kept.

In terms of the network system, UPDATE models the fact that intra-system communications take precedence over inter-system communication, since NEWIN will yield σ_{mB} containing the message buffers updated by the incoming inter-system messages and NEWOUT will yield in $\sigma_{messages}$ all the intra-system messages produced during phase 2.

(8) NEWIN: $\Sigma_{mB} \to \Sigma_{mB}$

NEWIN(σ_{mB}) $\equiv$ Compose(σ_{mB},REC{ })

Comment: The value of NEWIN is the current message buffers merged with the incoming external messages, so that incoming new messages overwrite old messages.

(9) COMPOSE: $\Sigma_{mB} \times \Sigma_{messages} \to \Sigma_{mB}$

COMPOSE(σ_{mB},$\sigma_{messages}$) $\equiv$ PRIMITIVE

Comment: Compose merges the message of σ_{mB} and $\sigma_{messages}$ with $\sigma_{messages}$ taking precedence.

(10) REC: $\phi \to \Sigma_{messages}$

REC({ }) $\equiv$ XCLOCREC$_i${ }

Comment: The i indicates that we are in network system i , XCLOCREC$_i$ receives the external messages.

(11) NEWOUT: $\Sigma_{IB} \to \Sigma_{messages}$

NEWOUT(σ_{IB}) $\equiv$ CONCAT(INSYS(σ_{IB}),SEND(OUTSYS(σ_{IB})))

Comment: NEWOUT inspects the interface buffers and sends the inter-system messages and carries forward the intra-system messages with messages for the same destination concatenated.

(12) CONCAT: $\Sigma_{IB} \to \Sigma_{IB}$

CONCAT(σ_{IB}) $\equiv$ PRIMITIVE, concatenates all messages for the same destination into one message and carries forward the others.

(13) INSYS: $\Sigma_{IB} \rightarrow \Sigma_{IB}$

INSYS(σ_{IB}) $\equiv$ INSYS$_{11}$(v_{IB})

Comment: INSYS is specified as a value function.

INSYS selects all the intra-system messages.

(13a) INSYS$_{11}$: $V_{IB} \rightarrow V_{IB}$

INSYS$_{11}$(v_{IB}) $\equiv$ PRIMITIVE,

Comment: Value of INSYS$_{11}$(v_{IB}) = v_{IB} if v_{IB} is to go to a destination in the system and ϕ otherwise.

(14) OUTSYS: $\Sigma_{IB} \rightarrow \Sigma_{IB}$

OUTSYS(σ_{IB}) $\equiv$ OUTSYS$_{11}$(v_{IB})

Comment: OUTSYS is specified as a value function.

OUTSYS selects all the inter-system messages.

(14a) OUTSYS$_{11}$: $V_{IB} \rightarrow V_{IB}$

OUTSYS$_{11}$(v_{IB}) $\equiv$ PRIMITIVE, value is v_{IB} if v_{IB} is to go to a destination outside the system, and ϕ otherwise.

(15) SEND: $\Sigma_{IB} \rightarrow \phi$

SEND(σ_{IB}) $\equiv$ XCLOCSEND$_i$(σ_{IB})

Comment: SEND transmits the external messages, the i indicates that our single network system is the ith system in the network.

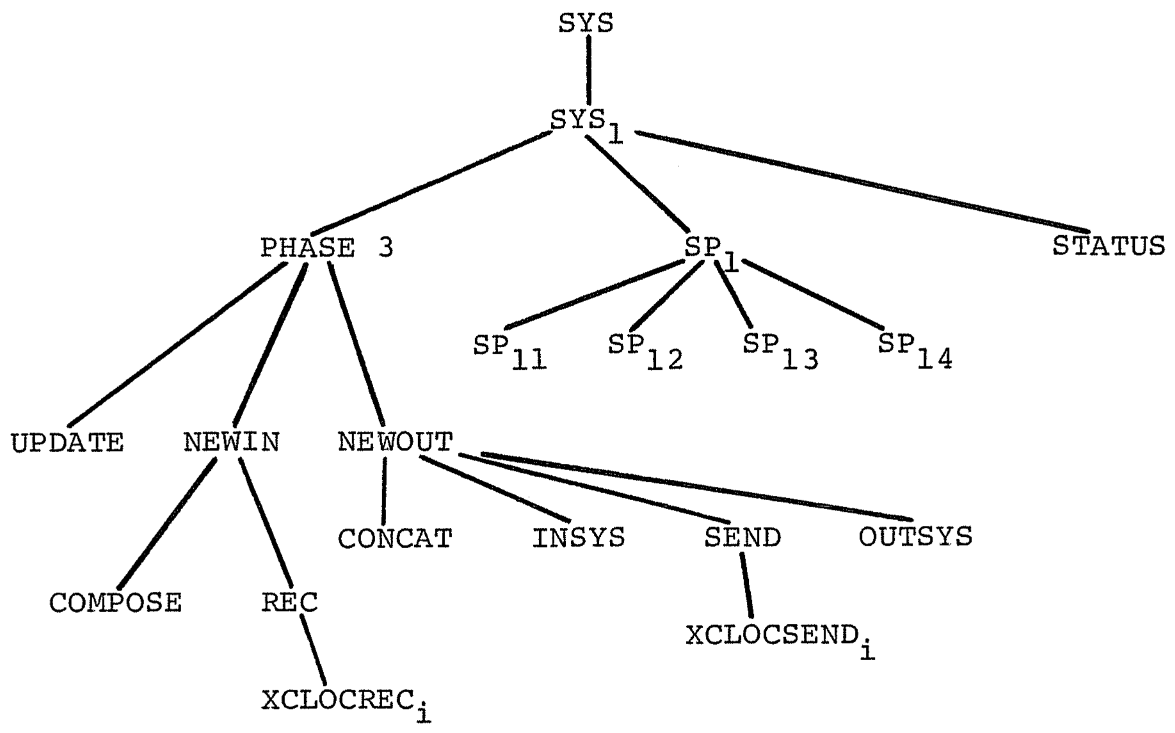


FIG. 5: Function tree for detailed specification of SYS.

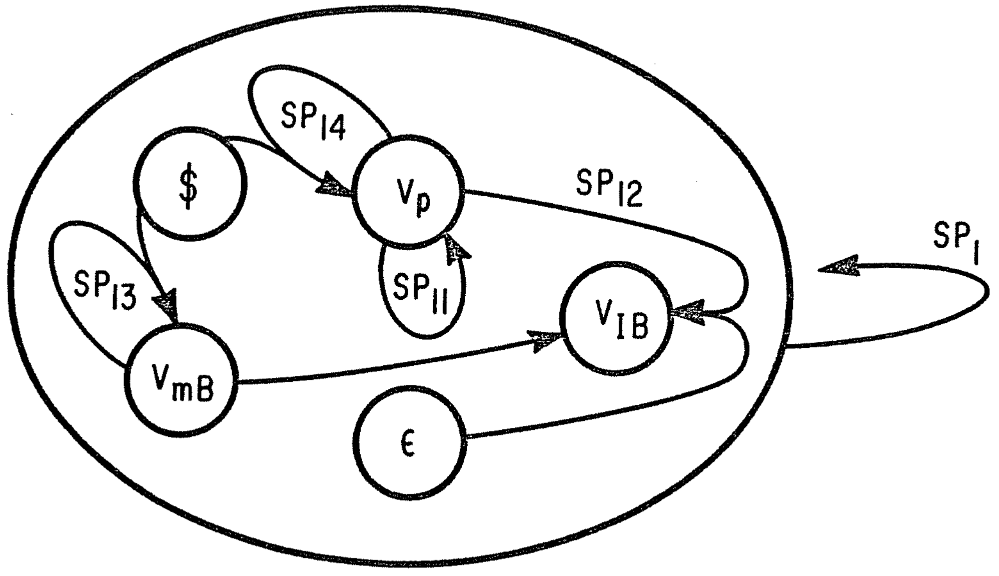


Fig. 6: Process graph for detailed specification of SP_1.

D.2.3.2 Detailed Specification of Realworld

(1) REALWORLD: $\Sigma_{REAL} \rightarrow \Sigma_{REAL}$

REALWORLD(σ_{REAL}) $\equiv$ REALWORLD$_1$($\sigma_1, \ldots, \sigma_N$)

(2) REALWORLD$_1$: $\Sigma_{TO_1} \times \ldots \times \Sigma_{TO_N} \rightarrow \Sigma_{TO_1} \times \ldots \times \Sigma_{TO_N}$

REALWORLD$_1$($\sigma_{TO_1}, \ldots, \sigma_{TO_N}$) $\equiv$ JOIN(PACKETIN( ),
TRYSEND($\sigma_{TO_1}, \ldots, \sigma_{TO_N}$))

Comment: REALWORLD$_1$ is component successor function.

(3) JOIN: $\Sigma_{NEW} \times \Sigma_{TO_1} \times \ldots \times \Sigma_{TO_N} \rightarrow \Sigma_{TO_1} \times \ldots \times \Sigma_{TO_N}$

JOIN($\sigma_{NEW}, \sigma_{TO_1}, \ldots, \sigma_{TO_N}$) $\equiv$ PRIMITIVE

Comment: JOIN updates the σ_j from σ_{NEW}.

(4) PACKETIN: $\phi \rightarrow \Sigma_{NEW}$

PACKETIN( ) $\equiv$ CHOICE(GREC$_1${ },...,GREC$_N${ })

Comment: PACKETIN receives messages transmitted from all systems and merges them into one packet with at most 1 message/destination.

(5) CHOICE: $\Sigma_{FROM_1} \times \ldots \times \Sigma_{FROM_N} \rightarrow \Sigma_{NEW}$

CHOICE($\sigma_{FROM_1}, \ldots, \sigma_{FROM_N}$) $\equiv$ PRIMITIVE

Comment: Merge the σ_{FROM_j} with the condition that at most 1 message/destination.

(6) GREC$_j$: $\phi \rightarrow \Sigma_{FROM_j}$

GREC$_j${ } $\equiv$ XSLOCSEND$_j${ }

Comment: $GREC_j\{\ \}$ receives a packet, (if any) transmitted from the jth system.

(7) TRYSEND: $\Sigma_{TO_i} \times \ldots \times \Sigma_{TO_N} \rightarrow \Sigma_{TO_1} \times \ldots \times \Sigma_{TO_N}$

$TRYSEND(\sigma_{TO_1},\ldots,\sigma_{TO_N}) \equiv (GSEND_1(\sigma_{TO_1}),\ldots,GSEND_N(\sigma_{TO_N}))$

Comment: TRYSEND tries to send the as yet undelivered packet for each system, to that system.

(8) $GSEND_i$: $\Sigma_{TO_i} \rightarrow \Sigma_{TO_i}$

$GSEND_i(\sigma_{TO_i}) \equiv XSLOCREC_i\{\sigma_{TO_i}\}$

Comment: $GSEND_i$ attempts to send to the ith system.

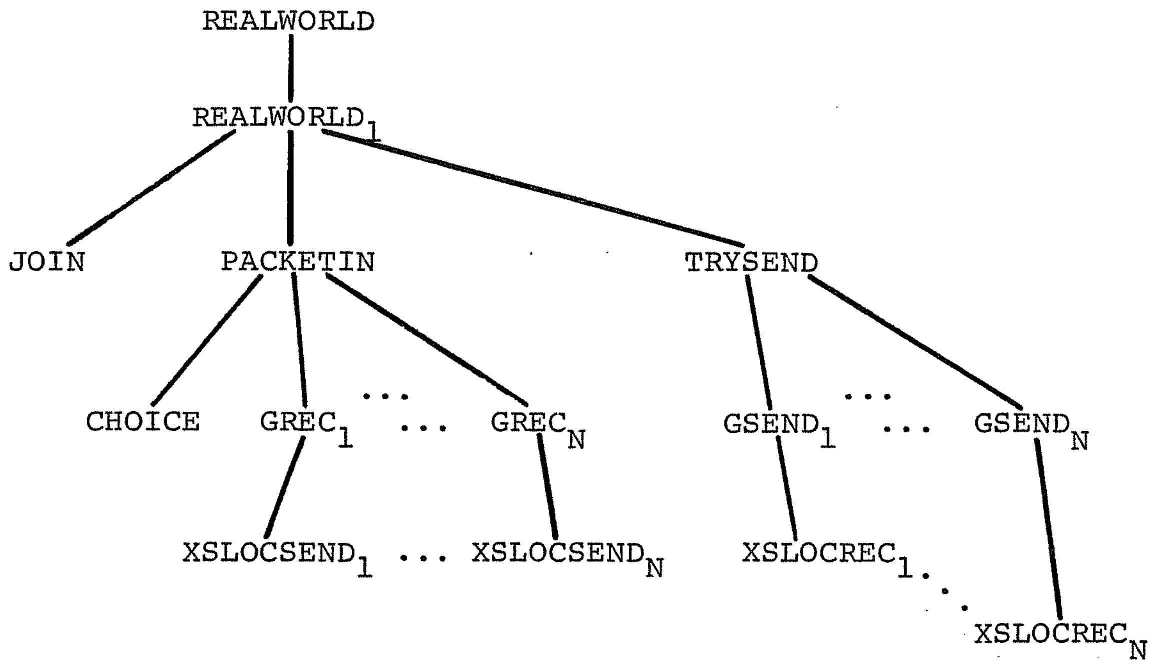


FIG. 7: Function definition tree for detailed specification of REALWORLD.

D.2.3.3 Function Table for Detailed Specification

TYPE	MAPPING	DEFINITION
State Successor	SYS: $\Sigma_{SYS} \rightarrow \Sigma_{SYS}$	SYS $\equiv$ SYS_1
Component	SYS_1: $\Sigma_p \times \Sigma_{mB} \rightarrow \Sigma_p \times \Sigma_{mB}$	$SYS_1(\sigma_p,\sigma_{mB}) \equiv$ PHASE 3(SP_1(STATUS(σ_p,σ_{mB})))
Component	STATUS: $\Sigma_p \times \Sigma_{mB} \rightarrow \Sigma_{\overline{p}} \times \Sigma_{\overline{mB}}$	PRIMITIVE
Component	SP_1: $\Sigma_{\overline{p}} \times \Sigma_{\overline{mB}} \rightarrow \Sigma_p \times \Sigma_{mB} \times \Sigma_{IB}$	$SP_1 \equiv SP_{11}, SP_{12}, SP_{13}, SP_{14}$
Value	SP_{11}: $V_p \times V_{mB} \rightarrow V_p \times V_{IB}$	PRIMITIVE
Value	SP_{12}: $V_p \times \{\varepsilon\} \rightarrow V_p \times V_{IB}$	PRIMITIVE
Value	SP_{13}: $\{\varepsilon\} \times V_{mB} \rightarrow V_p$	$SP_{13}(\$, v_{mB}) \equiv (t:(v_{mB}))$
Value	SP_{14}: $V_p \times \{\varepsilon\} \rightarrow V_p$	$SP_{14}(v_p, \$) \equiv (t:(v_{mB}))$
Component	Phase 3: $\Sigma_p \times \Sigma_{mB} \times \Sigma_{IB} \rightarrow \Sigma_p \times \Sigma_{mB}$	PHASE 3($\sigma_p,\sigma_{mB},\sigma_{IB}$) $\equiv$ (σ_p,UPDATE(NEWIN(σ_{mB}), NEWOUT(σ_{IB})))
Component	UPDATE: $\Sigma_{mB} \times \Sigma_{messages} \rightarrow \Sigma_{mB}$	PRIMITIVE
Component	NEWIN: $\Sigma_{mB} \rightarrow \Sigma_{mB}$	NEWIN(σ_{mB}) $\equiv$ COMPOSE(σ_{mB},REC{ })
Component	COMPOSE: $\Sigma_{mB} \times \Sigma_{messages} \rightarrow \Sigma_{mB}$	PRIMITIVE
Component	REC: $\phi \rightarrow \Sigma_{messages}$	REC({ }) $\equiv$ $XCLOCREC_i${ }
Component	NEWOUT: $\Sigma_{IB} \rightarrow \Sigma_{messages}$	NEWOUT(σ_{IB}) $\equiv$ CONCAT(INSYS(σ_{IB}), SEND(OUTSYS(σ_{IB})))

TYPE	MAPPING	DEFINITION
Component	CONCAT: $\Sigma_{IB} \rightarrow \Sigma_{IB}$	PRIMITIVE
Component	INSYS: $\Sigma_{IB} \rightarrow \Sigma_{IB}$	$\text{INSYS}(\sigma_{IB}) \equiv \text{INSYS}(v_{IB})$
Value	INSYS_{11}: $V_{IB} \rightarrow V_{IB}$	PRIMITIVE
Component	OUTSYS: $\Sigma_{IB} \rightarrow \Sigma_{IB}$	$\text{OUTSYS}(\sigma_{IB}) \equiv \text{OUTSYS}_{11}(v_{IB})$
Value	OUTSYS_{11}: $V_{IB} \rightarrow V_{IB}$	PRIMITIVE
Component	SEND: $\Sigma_{IB} \rightarrow \phi$	$\text{SEND}(\sigma_{IB}) \equiv \text{XCLOCSEND}_i(\sigma_{IB})$
State Successor	REALWORLD: $\Sigma_{REAL} \rightarrow \Sigma_{REAL}$	$\text{REALWORLD}(\sigma_{REAL}) \equiv \text{REALWORLD}_1(\sigma_{TO_1},\ldots,\sigma_{TO_N})$
Component	REALWORLD_1: $\prod_{i=1}^{N} \Sigma_{TO_i} \rightarrow \prod_{i=1}^{N} \Sigma_{TO_i}$	$\text{REALWORLD}_1(\sigma_{TO_1},\ldots,\sigma_{TO_N}) \equiv \text{JOIN}(\text{PACKETIN}(\ ),$ $\text{TRYSEND}(\sigma_{TO},\ldots,\sigma_{TO_N}))$
Component	JOIN: $\Sigma_{NEW} \times \prod_{i=1}^{N} \Sigma_{TO_i} \rightarrow \prod_{i=1}^{N} \Sigma_{TO_i}$	PRIMITIVE
Component	PACKETIN: $\phi \rightarrow \Sigma_{NEW}$	$\text{PACKETIN} \equiv \text{CHOICE}(\text{GREC}_1(\ ),\ldots,\text{GREC}_N(\ ))$
Component	CHOICE: $\prod_{i=1}^{N} \Sigma_{FROM_i} \rightarrow \prod_{i=1}^{N} \Sigma_{FROM_i}$	PRIMITIVE
Component	GREC_i: $\phi \rightarrow \Sigma_{FROM_i}$	$\text{GREC}_i(\ ) \equiv \text{XSLOCSEND}_i\{\ \}$
Component	TRYSEND: $\prod_{i=1}^{N} \Sigma_{TO_i} \rightarrow \prod_{i=1}^{N} \Sigma_{TO_i}$	$\text{TRYSEND}(\sigma_{TO},\ldots,\sigma_{TO_N}) \equiv (\text{GSEND}(\sigma_{TO_1}),\ldots,$ $\text{GSEND}(\sigma_{TO_N}))$

TYPE	MAPPING	DEFINITION
Component	$GSEND_1: \Sigma_{TO_1} \rightarrow \Sigma_{TO_i}$	$GSEND_i(\sigma_{TO_i}) \equiv XSLOCREC_i\{\sigma_{TO_i}\}$

EXCHANGES

$XSLOCREC_i(\ )$

$XCLOCREC_i(\ )$

$XSLOCSEND_i(\ )$

$XCLOCSEND_i(\ )$

INDEX