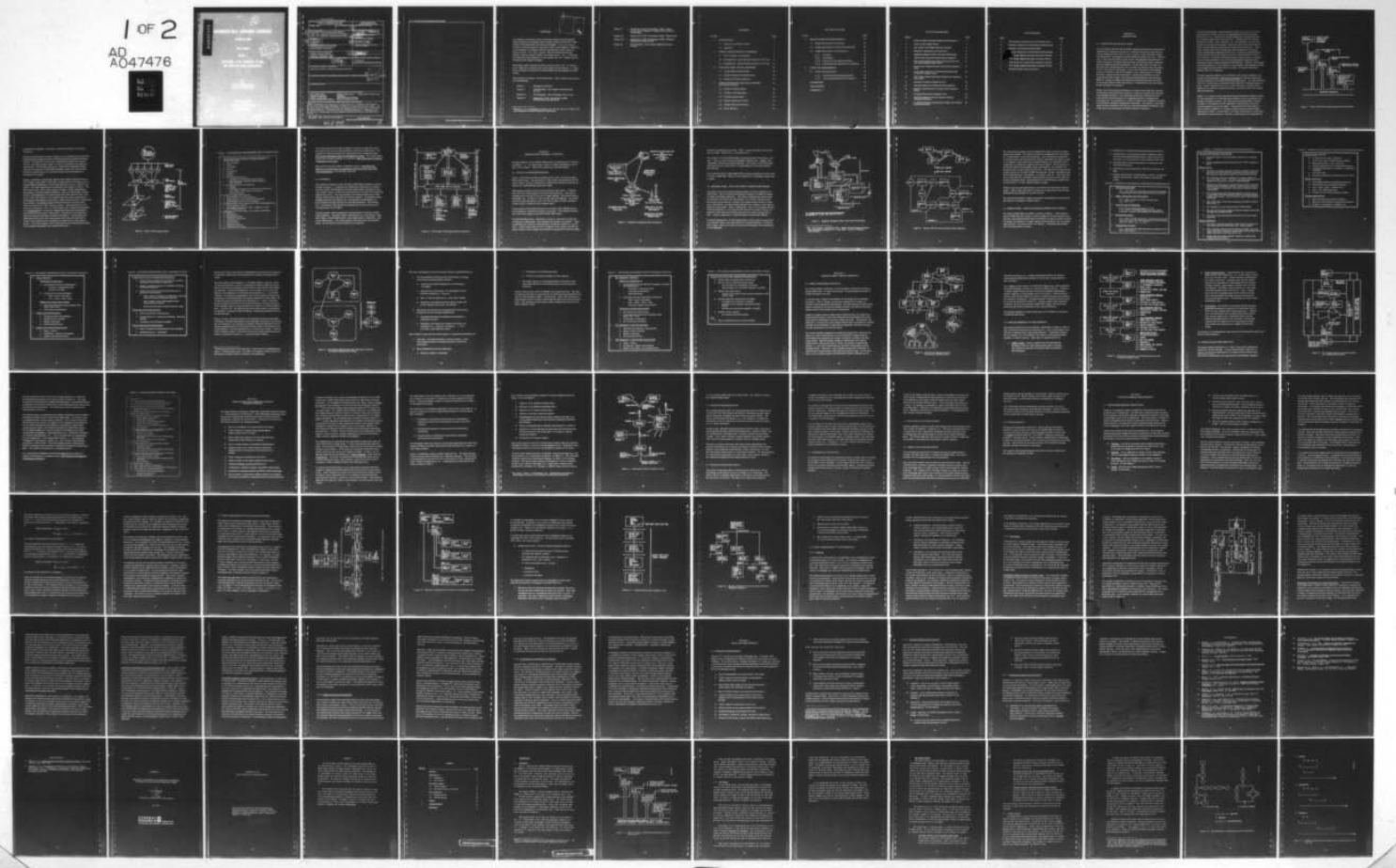
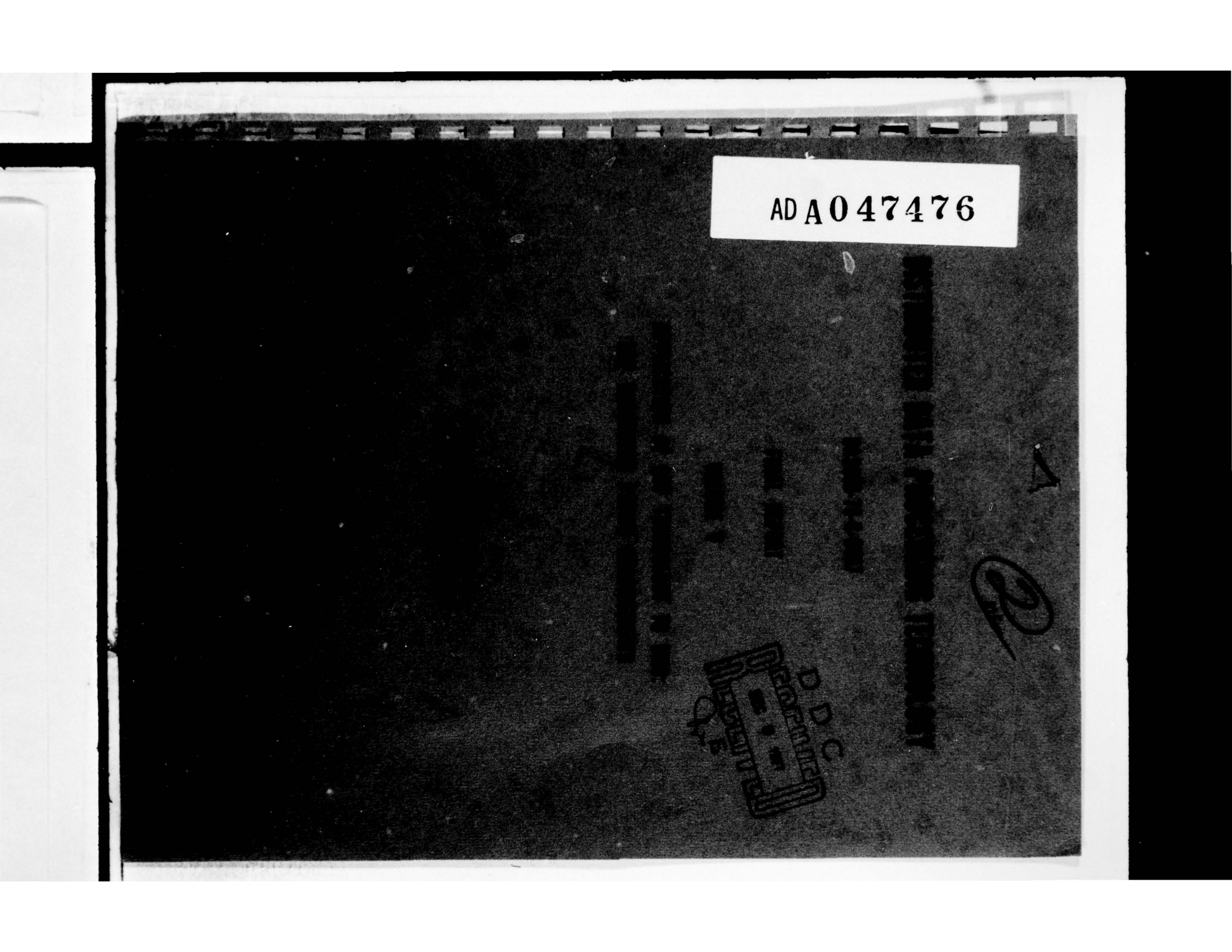
AD-A047 476 HONEYWELL INC MINNEAPOLIS MINN SYSTEMS AND RESEARCH --ETC F/G 9/2
DISTRIBUTED DATA PROCESSING TECHNOLOGY. VOLUME V. APPLICATION O--ETC(U)
SEP 77 D PALMER, E BALKOVICH, R WOOD DASG60-76-C-0087
UNCLASSIFIED 77SRC69 NL

1 OF 2
AD A047476




AD A047476

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (WHEN DATA ENTERED)

REPORT DOCUMENTATION PAGE	READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER	2. GOV'T ACCESSION NUMBER
3. RECIPIENT'S CATALOG NUMBER	
4. TITLE (AND SUBTITLE) (6) Distributed Data Processing Technology. Volume V. Application of DDP Technology to BMD: DDP Subsystem Design Requirements.	5. TYPE OF REPORT/PERIOD COVERED (9) Final Report, October 1976 to October 1977,
6. PERFORMING ORG. REPORT NUMBER (14) 77SRC69	
7. AUTHOR(S) (10) D. Palmer, E. Balkovich, R. Wood	8. CONTRACT OR GRANT NUMBER(S) (15) DASG60-76-C-0087
9. PERFORMING ORGANIZATIONS NAME/ADDRESS General Research Corporation Santa Barbara, California	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME/ADDRESS Ballistic Missile Defense Advanced Technology Center Huntsville, Alabama 35807	12. REPORT DATE (11) September 1977
	13. NUMBER OF PAGES 112
14. MONITORING AGENCY NAME/ADDRESS (IF DIFFERENT FROM CONT. OFF.) (12) 114p.	15. SECURITY CLASSIFICATION (OF THIS REPORT) Unclassified
	15a. DECLASSIFICATION DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (OF THIS REPORT)

Approved for public release; distribution unlimited.

DDC RECEIVED DEC 8 1977 F

17. DISTRIBUTION STATEMENT (OF THE ABSTRACT ENTERED IN BLOCK 20, IF DIFFERENT FROM REPORT)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (CONTINUE ON REVERSE SIDE IF NECESSARY AND IDENTIFY BY BLOCK NUMBER)

DDP design theory
Boundary conditions
System development
Design engineering modeling
Design requirements
Monitors
SAFEGUARD partitioning
Baseline design activities
Performance requirements

20. ABSTRACT (CONTINUE ON REVERSE SIDE IF NECESSARY AND IDENTIFY BY BLOCK NUMBER)

The development and examination of requirements for a subsystem level "baseline theory" for DDP design technology for BMD are the major topics of this report. The main purpose of the baseline theory was to help identify critical issues and needs for further research. The baseline theory, although incomplete as a design tool, also provides a starting point for future design technology development and experimentation.

HD-169 REV 11/74

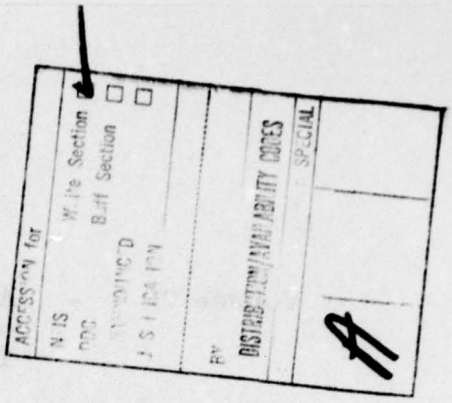
DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (WHEN DATA ENTERED)

402 349

LB



FOREWORD

The research documented in this volume was conducted under Ballistic Missile Advanced Technology Center contract number DASG60-76-C-0087, entitled "Distributed Data Processing Technology." The work was performed by General Research Corporation (GRC), Santa Barbara, California as subcontractor to Honeywell Systems and Research Center under the direction of Mr. C. R. Vick, Director, Data Processing Directorate, Ballistic Missile Defense Advanced Technology Center. Mr. J. Scalf was the BMDATC project engineer for this contract; Ms. B. C. Stewart was the Honeywell/GRC program manager.

This report covers work from October 1976 to October 1977. D. Palmer and E. Balkovich, both from GRC, participated in this research. R. Wood of the University of California at Santa Barbara also participated on behalf of GRC.

This document is Volume V of the final report. Other volumes of the report are the following:*

Volume I	-	Management Summary
Volume II	-	DDP Rationale: The Program Planning Point of View
Volume III	-	DDP Rationale: The Technology Point of View
Volume IV	-	Application of DDP Technology to BMD: Architectures and Algorithms

*Volumes V, VI, the appendix to Volume VII, and one section of Volume VIII were prepared by General Research Corporation.

Volume VI	**-**	**Application of DDP Technology to BMD: Impact on Current DP Subsystem Design and Development Technologies**
Volume VII	**-**	**Application of DDP Technology to BMD: Experiments**
Volume VIII	**-**	**Application of DDP Technology to BMD: Research Performance Measurement**
Volume IX	**-**	**DDP Rationale: The Program Experience Point of View**

CONTENTS

Section		Page
1	INTRODUCTION	1
	1.1 Objectives and Scope of Work	1
	1.2 Approach	7
2	DESIGN THEORY BOUNDARY CONDITIONS	9
	2.1 Place in System Development	9
	2.2 Starting Point: Input Document Contents and Format	11
	2.3 Ending Point: Output Document Contents and Format	14
3	BASELINE DESIGN THEORY DEFINITION	26
	3.1 Design Engineering Modeling	26
	3.2 Baseline Sequence of Requirements	28
	3.3 Baseline Activities Definition	30
4	DESIGN TECHNOLOGY RESEARCH OVERVIEW AND EXPECTATIONS	34
	4.1 Problem Definition Space	39
	4.2 Problem Technology Space	39
	4.3 Feasible Solution Space	40
	4.4 Solution Evaluation Criteria	41
	4.5 Design Evolution Procedure	41
	4.6 Other Elements	42

CONTENTS (concluded)

Section			Page
5	SPECIFIC RESEARCH REQUIREMENTS		43
	5.1 Design Requirements Translation		43
	5.2 Design Requirements Analysis/Decomposition		48
	5.3 Design Requirements Transformation		54
		5.3.1 Overview	54
		5.3.2 Parallelism	56
		5.3.3 Control and Data Base Formalism	63
		5.3.4 Partitioning and Distribution of Capability	65
6	ISSUES AND REQUIREMENTS		67
	6.1 Research Requirements		67
		6.1.1 Structures Requirements Research	69
		6.1.2 Unstructured Requirements Research	70
	REFERENCES		72
	BIBLIOGRAPHY		74
	APPENDIX A		75

LIST OF ILLUSTRATIONS

Figure		Page
1	Model of Data Processing Design and Development	3
2	Scope of DDP Design Theory	5
3	DDP Design Technology Research Approach	8
4	Safeguard Components and Contractors	10
5	Suggested Safeguard MDC Functional Partitioning	12
6	Typical DPE Structured Requirements Diagrams	13
7	DDP Design Engineering Output Document Overview: Functional Requirements Portion	21
8	Hierarchical Design Structure and Desired Characteristics	27
9	Preliminary Sequence of Addressing Requirements in Gaseline Design Theory	29
10	DDP Design Engineering Model Adopted as a Research Framework	31
11	Elements of Generic Design Activity	38
12	Example Requirement Definition Decomposition	49
13	Example Decomposition of Certain Growth Requirements	50
14	A Design Requirement Response Tree	52
15	Example Response Tree for Growth of Threat Component Numbers	53
16	A Parallel Pipeline Architecture for Real-Time Speech Feature Extraction	58

LIST OF TABLES

Table		Page
1	DDP Research Requirements and Considerations	6
2	Example Categories of Performance Requirements	15
3	Example Unstructured Requirements	16
4	Example Categories of Unstructured Requirements	17
5	DDP Design Engineering Input Document Contents	18
	DDP Design Engineering Input Document Format	19
7	DDP Subsystem Design Output Document Contents	24
8	DDP Design Engineering Output Document Format	25
9	Baseline Design Theory Activities	33

SECTION 1
INTRODUCTION

1.1 OBJECTIVES AND SCOPE OF WORK

Current Ballistic Missile Defense (BMD) requirements exceed the performance capabilities of centralized computers, necessitate excessively long data processing deployment lead times (5 years or more), create extremely complex testing problems, and change frequently during the system life cycle. Furthermore, many advanced BMD concepts involve the geographical distribution of system resources, including data processing, and thereby further complicate data processing design and development. On the other hand, the ability to distribute data processing tasks over a network of computers has been shown to offer many potential advantages, such as increased capacity (throughput), increased reliability, shorter development cycle, better growth (or change) facilities, reduced system vulnerability, architectural flexibility (to meet requirements more simply), and improved programmability and testability. However, to gain the inherent distributed processing advantages and satisfy the inherently difficult BMD requirements, a comprehensive data-processing design technology is necessary.

Another type of processing distribution is currently impacting many systems. Processing is being distributed locally (at a node) over numbers of micro and minicomputing elements. The computing architecture flexibility thus provided and the enormous advances in computing hardware technology now occurring (e.g., the computer on a "chip" via large-scale integrated circuitry) are drastically changing computational constraints. For instance, many tasks implemented by software in the past are now implementable in hardware, providing large speed advantages and better control of reliability.

Limited distributed data processing (DDP) design research has been conducted within the BMD community for (geographically) local processing distributions (e.g., PAR and Site Defense Module/Unit configuration) and for computer architecture distribution (e.g., SAFEGUARD CLC and Site Defense CDC 7700). Other research, e.g., for ARPANET, has been concerned with the sharing of computer resources rather than the sharing of a single, overall objective (meeting BMD requirements effectively). These early attempts to use DDP concepts have exposed problems in the basic theory of distributed computing systems (e.g., problems of distribution of control and data base) which must be solved if the full potential of DDP is to be realized. In addition, distributed defense concepts require a greater scope of research because of the need for geographical dispersion while solving the system-wide (homogeneous) BMD problem. BMDATC consequently initiated research in FY 77 to obtain the many potential DDP payoffs for BMD and to support advanced BMD system concepts. This report describes a portion of the FY 77 research.

The main long-term objective of the BMDATC DDP research program is to develop and demonstrate a DDP design technology for efficient implementation and validation of BMD data-processing requirements. The main objectives of the FY 77 effort were to demonstrate the feasibility of attaining long-term goals, identify important issues to be resolved by future research, define longer-term research and experiment plans, and develop initial DDP design data and guidelines.

Figure 1 is a model of a data-processing design development sequence useful for describing the FY 77 work and future requirements (to help define terminology and ranges of effort). The data-processing design phase is shown in Figure 1 to be applied to BMD "application requirements" which have been decomposed in an earlier phase. (Research of this earlier phase is being initiated by BMDATC in a program called Axiomatic Requirements Engineering [ARE].) Furthermore, DP design is followed by an implementation phase in which software, firmware, and hardware are constructed (or

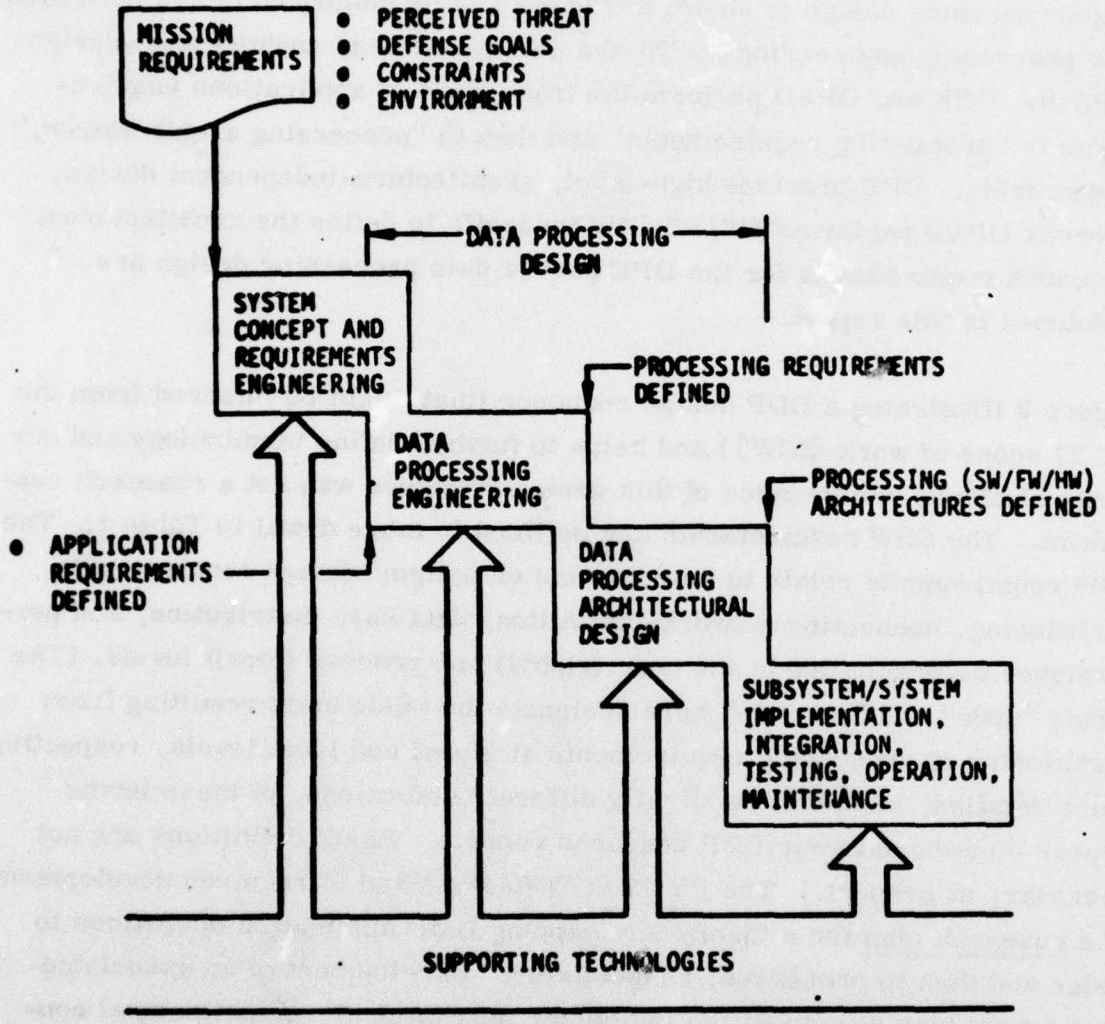


Figure 1. Model of Data Processing Design and Development

procured) and integrated. (Of course, iteration of activities is assumed necessary.)

Data-processing design is shown in Figure 1 to be composed of two activities: data processing engineering (DPE) and data processing architectural design (DPAD). DPE and DPAD perform the transitions of applications requirements to "processing requirements" and then to "processing architectures," respectively. DPE provides high-level, architecture-independent design, whereas DPAD performs SW/FW/HW tradeoffs to define the architectures. Research requirements for the DPE part of data processing design are developed in this report.

Figure 2 illustrates a DDP design sequence (that might be inferred from the FY 77 scope of work [SOW]) and helps to further define terminology and our range of effort; maintenance of this design structure was not a research constraint. The SOW requirements are outlined in more detail in Table 1. The SOW requirements relate to four aspects of design: design requirements partitioning, connecting networks definition, data base distribution, and performance determination at the node (global) and process (local) levels. (The terms "node" and "process" here designate the basic units resulting from partitioning of processing requirements at global and local levels, respectively; more detailed, and perhaps slightly different definitions, of these terms appear elsewhere in our DDP research reports. Exact definitions are not necessary at present.) The FY 77 SOW (tasks 4 and 5) required development of a research plan for a theory for mapping BMD application operations to nodes and then to processes, respectively. Development of an associated experiment plan was specified in another task (task 8). Architectural considerations (of the lower level design phase) were assessed under task 6. (Tasks 1 and 2 addressed the architectural considerations in less general terms.) The mutual impact of DDP requirements on existing data processing design support technologies were studied under task 7.

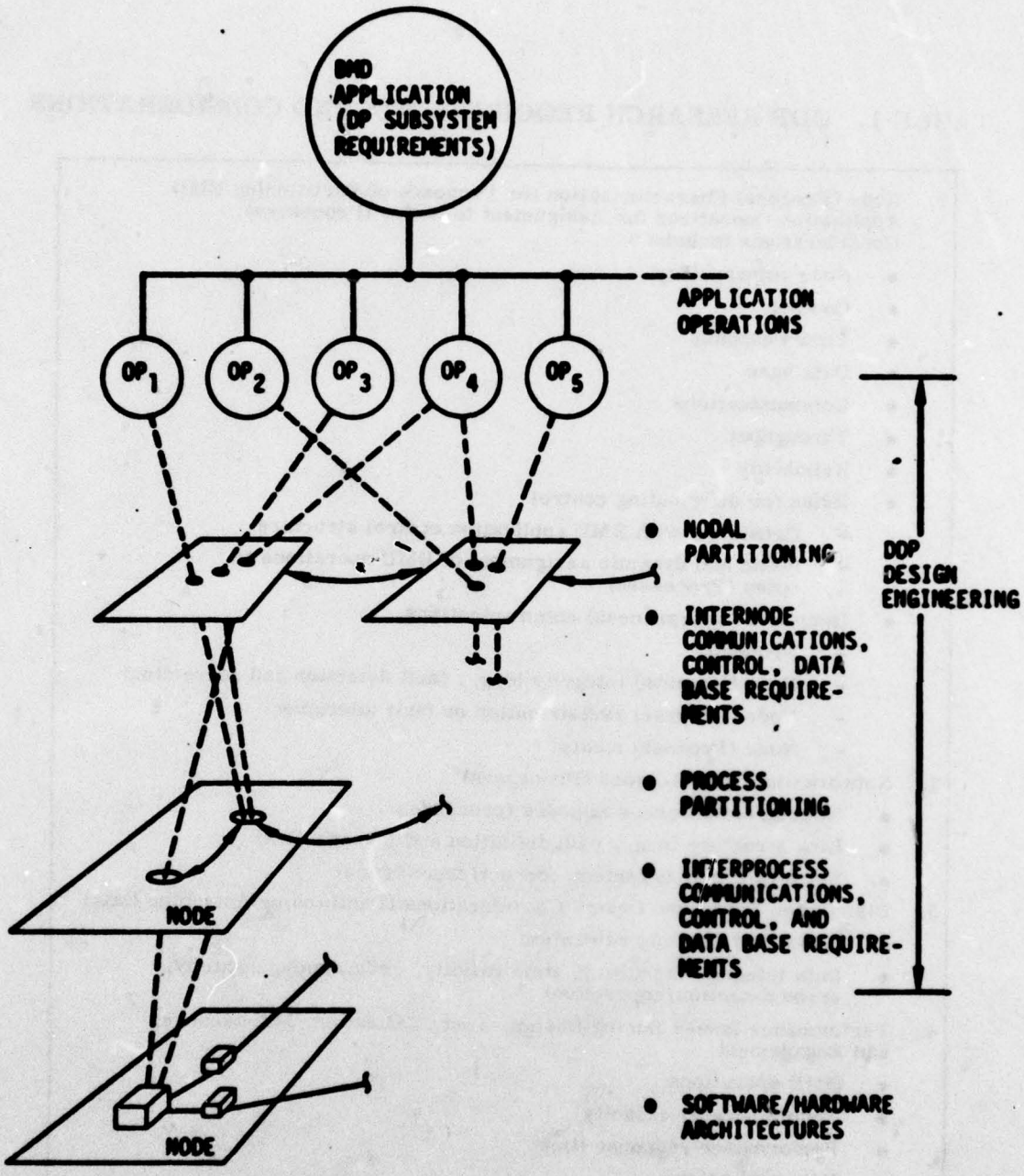


Figure 2. Scope of DDP Design Theory

TABLE 1. DDP RESEARCH REQUIREMENTS AND CONSIDERATIONS

1. Node (Process) Characterization for Purposes of Partitioning BMD Application Operations for Assignment to Nodes (Processes). Considerations Include:
   - Node vulnerability
   - Control
   - Time response
   - Data base
   - Communications
   - Throughput
   - Reliability
   - Rules for distributing control
     - Consistent with BMD application control structure
     - Static and dynamic assignment of BMD operations to notes (Processes)
   - Internode (Interprocess) communications
     - Protocol
     - Node (Process) integrity (e.g., fault detection and correction)
     - Node (Process) redistribution or fault tolerance
     - Node (Process) sanity
2. Networks to Connect Nodes (Processes)
   - Structure/Resources at nodes (processes)
   - Link structure (e.g., path definition and management)
   - Network integrity (detect/correct/reconfigure)
3. Distributed Data Base Design Considerations (Partitioning/Assigning Data)
   - Initial and dynamic relocation
   - Data integrity (accuracy, time validity, redundancy, security, error detection/correction)
4. Performance Issues During Design, Test, Validation, Maintenance, and Engagement
   - BMD operations
   - Node (process) capacity
   - Performance response time
   - Communications
   - Node vulnerability
   - Reliability/Availability
   - Validation and sets concepts

This report describes the tasks 4 through 6 research to define the issues and requirements basic to planning future research and experiments. The research and experiment plans are consolidated into a companion report, DDP Design Technology Research and Experiment Plans. The mutual impact of DDP and existing technologies is reported in DDP Design and Development Technologies Assessment.

More detail on our DDP design technology is given in The Role of the Monitor as an Abstraction in Designing the Control for Real-Time Data Processing Systems, GRC IM-2124, July 1977.

1.2 APPROACH

Our approach to laying the basis for future DDP design technology research is shown in Figure 3. It consists of a preliminary study phase, in which a "baseline theory" and associated research requirements are defined, and a "case studies" phase, in which the baseline theory is assessed for interesting applications (cases). The main purpose of the baseline theory was to help identify critical issues and needs for further research. The baseline theory, although incomplete as a design tool, also provides a starting point for future design technology development and experimentation. The development, examination, and anticipated extensions of the baseline theory are the major topics of this report.

The case studies involved applying the baseline theory to portions of BMD design problems. Particular efforts concentrated on an earth-based, multistatic radar, terminal BMD system, and a space-based, trilateration, midcourse BMD system. The results appear in the final form of the baseline theory, but the details of these studies are not reported.

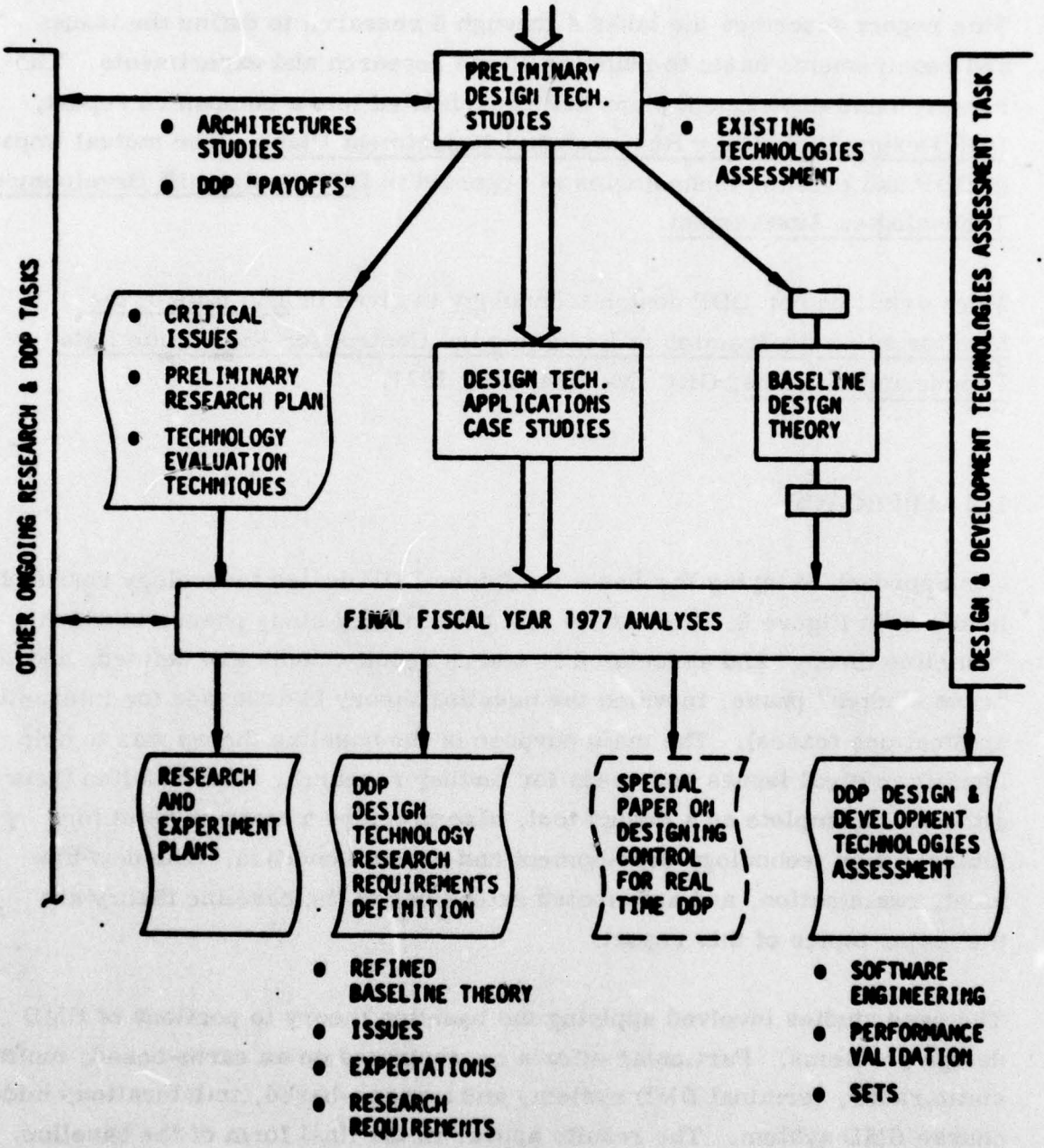


Figure 3. DDP Design Technology Research Approach

SECTION 2
DESIGN THEORY BOUNDARY CONDITIONS

We begin definition of the baseline design theory by identifying the "boundary conditions," i.e., the design theory's place in system development, its inputs, and its outputs. These three features are described below.

2.1 PLACE IN SYSTEM DEVELOPMENT

Figure 1 illustrates the system development life cycle in which the design theory will be used. The design theory specifically must support the second step of Figure 1 which is preceded by a requirements definition step and followed by an implementation specification step.

BMD systems usually do not have the simple life cycle shown. A BMD system life cycle more typically is iterative and evolutionary, with partial developments occurring in the form of "paper studies," test range versions, prototypes, etc., in advance of the primary life cycle depicted. At present, we will consider the primary life cycle, but our theory will be maintained robust enough to handle the more complex cycle.

The boundaries of the steps are shown as not being firmly defined because of the necessary overlap and iteration of activities. This overlap is discussed further with respect to the SAFEGUARD BMD System.

The SAFEGUARD System, illustrated in Figure 4, shows the conventional partitioning of BMD systems into hardware and software components. These components were made the responsibilities of separate contractors, as indicated. If DDP design theory were to be applied in this context, the DDP designer would be free to assign requirements to geographical locations (nodes)

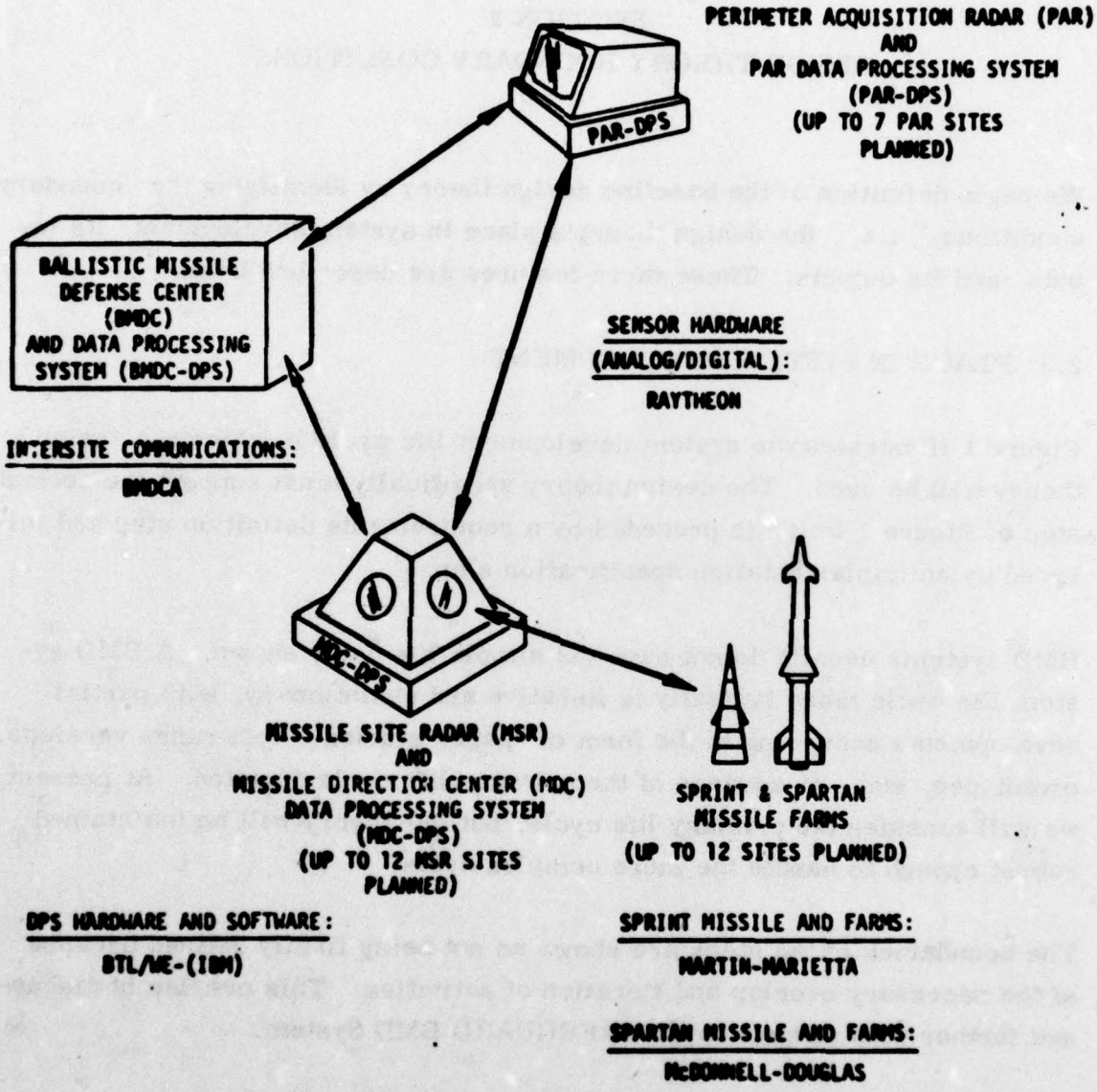


Figure 4. Safeguard Components and Contractors

and then to processes at the nodes. (Here, a node is defined to be the location of a radar, missile farm, or defense center.)

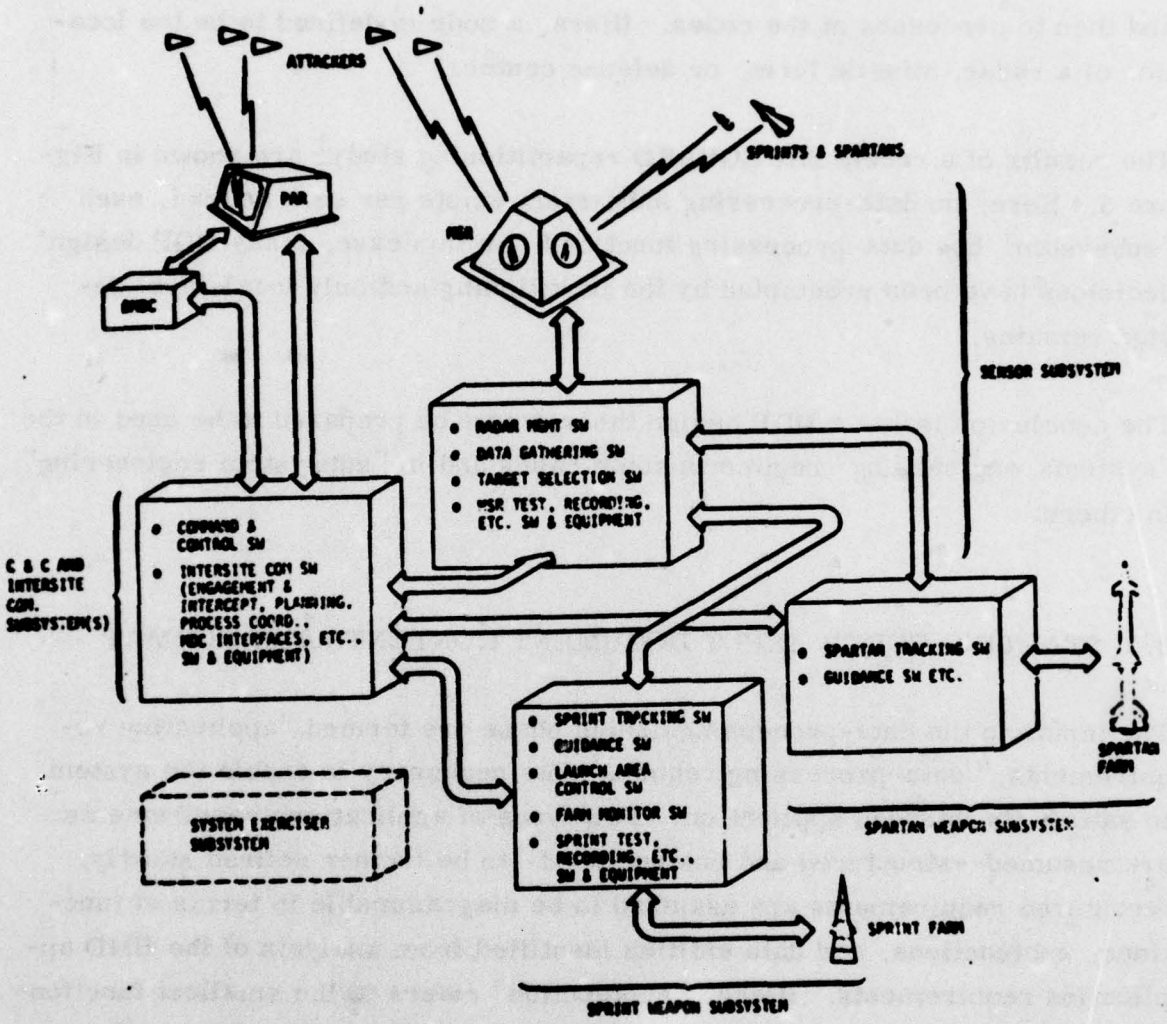
The results of a recent SAFEGUARD repartitioning study[1] are shown in Figure 5. Here, no data-processing subsystem exists per se. Instead, each "subsystem" has data-processing functions. In this case, many DDP design decisions have been preempted by the partitioning and only local-level design remains.

The conclusion is that a DDP design theory must be prepared to be used in the "systems engineering" regime in some cases and in "subsystem engineering" in others.

2.2 STARTING POINT: INPUT DOCUMENT CONTENTS AND FORMAT

The inputs to the data-processing design phase are termed "application requirements," data-processing requirements necessary to enable the system to satisfy its intended application. Two types of applications requirements are assumed--structured and unstructured--to be further defined shortly. Structured requirements are assumed to be diagrammable in terms of functions, subfunctions, and data entities identified from analysis of the BMD application requirements. (Here, "subfunction" refers to the smallest functional element identified.) The diagrams are assumed to indicate "natural" sequences of data transformations as seen from the system concept point of view. Figure 6 illustrates a typical structured requirements diagram.

In specific applications, the DDP design theory must be able to accept either more detailed inputs--with associated loss of design flexibility--or less detailed inputs--which could likely produce deficient designs if the deficiencies are not identified.



*REF: BTL REPORT, FEASIBILITY STUDY - IMPACT OF PARTITIONING ON LARGE-SCALE SYSTEMS DEVELOPMENT (U) - CASE 28487, HAYDEN & SUNSHINE, 5 MAY 1976 (S)

Figure 5. Suggested Safeguard MDC Functional Partitioning*

*Ref: BTL Report, Feasibility Study - Impact of Partitioning on Large-Scale Systems Development (U) - Case 28487, Hayden & Sunshine, 5 May 1976 (S)

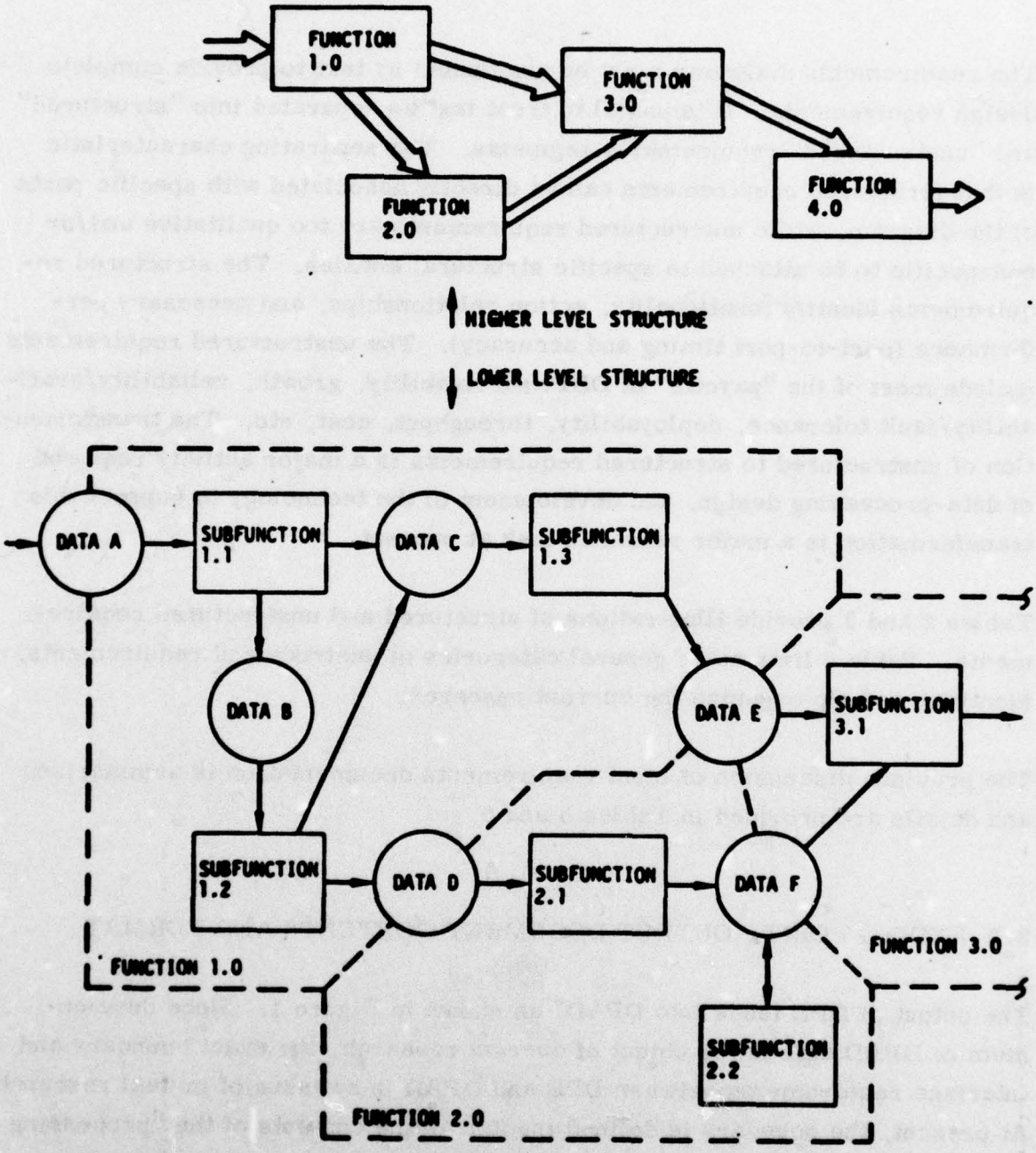


Figure 6. Typical DPE Structured Requirements Diagrams

The requirements diagrams must be augmented by text to provide complete design requirements. It is useful to treat text as separated into "structured" and "unstructured" requirements segments. The separating characteristic is that structured requirements can be directly associated with specific parts of the diagram, while unstructured requirements are too qualitative and/or nonspecific to be attached to specific structural entities. The structured requirements identify functionality, action relationships, and necessary performance (port-to-port timing and accuracy). The unstructured requirements include most of the "payoffs" of DDP: survivability, growth, reliability/availability/fault tolerance, deployability, throughput, cost, etc. The transformation of unstructured to structured requirements is a major activity required of data-processing design, and development of the technology to support this transformation is a major research task at present.

Tables 2 and 3 provide illustrations of structured and unstructured requirements. Table 4 lists more general categories of unstructured requirements, identified to help organize the current research.

The previous discussion of input requirements documentation is summarized and details are provided in Tables 5 and 6.

2.3 ENDING POINT: OUTPUT DOCUMENT CONTENTS AND FORMAT

The output of DPE feeds into DPAD, as shown in Figure 1. Since development of DPAD also is the object of current research, the exact boundary and interface requirements between DPE and DPAD is an issue of mutual research. At present, the boundary is defined insofar as the contents of the "processing requirements," the output of DPE, are defined. Processing requirements are currently defined to contain:

1.) Identification of processing nodes (geographical locations)

2.) Identification of "processing structures" (abstract requirement structures denoting functionality, data, control, performance, and communications requirements) at each node.

3.) Architecturally-important attributes of components of the processing structures.

4.) Unstructured requirements that could not be treated during DPE.

5.) Testing requirements for DPAD (plans, criteria, simulations, support requirements, mission and system requirements traces)

TABLE 2. EXAMPLE CATEGORIES OF PERFORMANCE REQUIREMENTS

1. Thruput/Capacity/Size

   E.G., The DPS must satisfactorily maintain at least N simultaneous tracks under (given) scenario conditions

2. System Timing (Port-To-Port Timing)

   E.G., Search return to verify command time, T_{sv} = xx.x ± ms

3. Function Event Probabilities

   Form: Prob (event | condition(s)) ≥ 0.95
   E.G., Event = Bulk filter gates will contain object
   Condition = Object is RV targeted into defended area

4. Subfunction Accuracy

   E.G., After at least one second of continuous track with The S/N > 10 dB, the predicted range r_p from the track filter will satisfy $|r_p - r_t| \leq X$ m

5. Computational Accuracy

   E.G., Seven decimal place precision is required in face switch calculations

TABLE 3. EXAMPLE UNSTRUCTURED REQUIREMENTS

Allocated DP Subsystem Requirements

- Less than XXX% of the RVs shall be lost while in end game track.
- Battle management shall ensure survival of XXX minuteman silos.

Design Criteria

- The data processing subsystem shall be capable of unmanned operation in a fully automatic mode after remote activation. Such operation shall be for XXX hours within XXX minutes.
- The number of silos to be defended in a single module may vary from XXX to XXX. The data-processing subsystem shall support this variation in number of silos defended.
- Subsystem growth against evolving threats shall be achieved without major subsystem redesign, without major impact on other subsystem components, and without extensive system inoperability.
- The communication net shall be designed so that the loss of a single data processing element will not seriously degrade system performance. The net shall be secure and protected from EMP.
- The vulnerability of the data-processing subsystem shall be XXX lb/sq. in.
- The reliability of each data processor shall be XXX.
- The data-processing subsystem shall be available within XXX minutes with probability XXX.
- The data processing subsystem shall have a 10-year operational life.

Design Constraints

- The data-processing subsystem shall include a computer at each radar site and at each farm site, and no others.
- The computers shall be interconnected such that, as a minimum, each radar computer communicates with the other three radar computers in the same group, etc.
- Radars and interceptors shall be capable of control from multiple-data processors.

TABLE 4. EXAMPLE CATEGORIES OF UNSTRUCTURED REQUIREMENTS

Design Criteria (Payoffs)

- Survivability (for loss of platform)
- Growth (for mission/system/threat changes)
- Deployability (time, risk, simplicity)
- Graceful degradation (for unanticipated attacks)
- Availability/Reliability (redundancy, fault/failure response)
- Life cycle cost
- Hardware utilization (or performance/cost ratio)

Design Constraints

- Permissible platforms (geographic locations)
- Permissible communications (channels)
- Size, weight, power, hardness, etc.
- Cost, risk, deployment time
- Thruput (if type of DP hardware is known)

Environmental Factors

- Data and control source/sink locations
- Data and control rates at source/sink interfaces
- Physical - temperature, radiation, etc.

TABLE 5. DDP DESIGN ENGINEERING INPUT DOCUMENT CONTENTS

1. System Definition

   Substructure Identification

   - Components/Partitions ("subsystems")
     - Functions, subfunctions ident.
     - Locations (not DPS)
   - Connecting structures/interfaces
     - Data, control, test paths
     - Com. modes, media, rates

   Substructure Design Requirements

   - Functional descriptions and operating rules (what, when, how)
   - Performance (quantitative)

2. System-Level Requirements

   - Allocated system requirements
   - Design Criteria (payoffs)
   - Design constraints
   - Environmental factors

3. System-Level Testing Requirements

   - Plans, criteria
   - Simulations, support requirements
   - Mission requirements traces

TABLE 6. DDP DESIGN ENGINEERING INPUT DOCUMENT FORMAT

Graphical (Or Formal Language) Representations

- Identify functions/subfunctions necessary to define functional requirements of application
- Identify geographical locations (hopefully excluding DP function locations)
- Identify connecting structures/interfaces due to application requirements
  - Show "natural" sequences of application functions without defining processing procedures
  - Show "natural" data relationships without defining data structures
  - Show branching without explicit decision nodes

Text of Structured Requirements

- Keyed to graphical representations
- Standardized form, machine processable, restricted English
- Consistency and traceability testable

Text of Unstructured Requirements

- Keyed to graphical representations where possible
- Standardized form, "parseable"

This section further describes the representational mechanisms and associated document contents and formats proposed for stating processing requirements.

The many design requirements (e.g., data base integrity, security, sanity, and executive definition) and the current state of abstract design techniques lead to representing processing requirements by functional and data abstraction elements. Functional abstractions, in our context, are no more than high-level representations of functional requirements; e.g., a box labeled "tracker" is an abstract representation of a not-yet-designed process which takes sensor measurement data and produces estimates of object states. Data abstractions are representations of resources (i.e., data or hardware) and permissible procedures for use of the resources, including procedures which regulate access to resources shared by a number of functional abstractions. In proper use, data abstractions can be used to enforce maintenance of modularity, suppress procedural details, and separate sequential and concurrent elements.

Figure 7 shows a structural diagram of the abstractions being investigated. Processes, functional abstractions which potentially can execute concurrently (per our definition), are composed of modules, functional abstractions which execute sequentially (but may have internal concurrency to be defined during implementation). The data abstractions are monitors and classes. Monitors, when completely defined, will contain the access privilege specifications necessary for executive function (e.g., operating system) design. *

*Monitors are used here as an abstraction, rather than as an implementation device. At lower design levels, the effect of an abstract monitor may be implemented in whatever manner is deemed most appropriate.

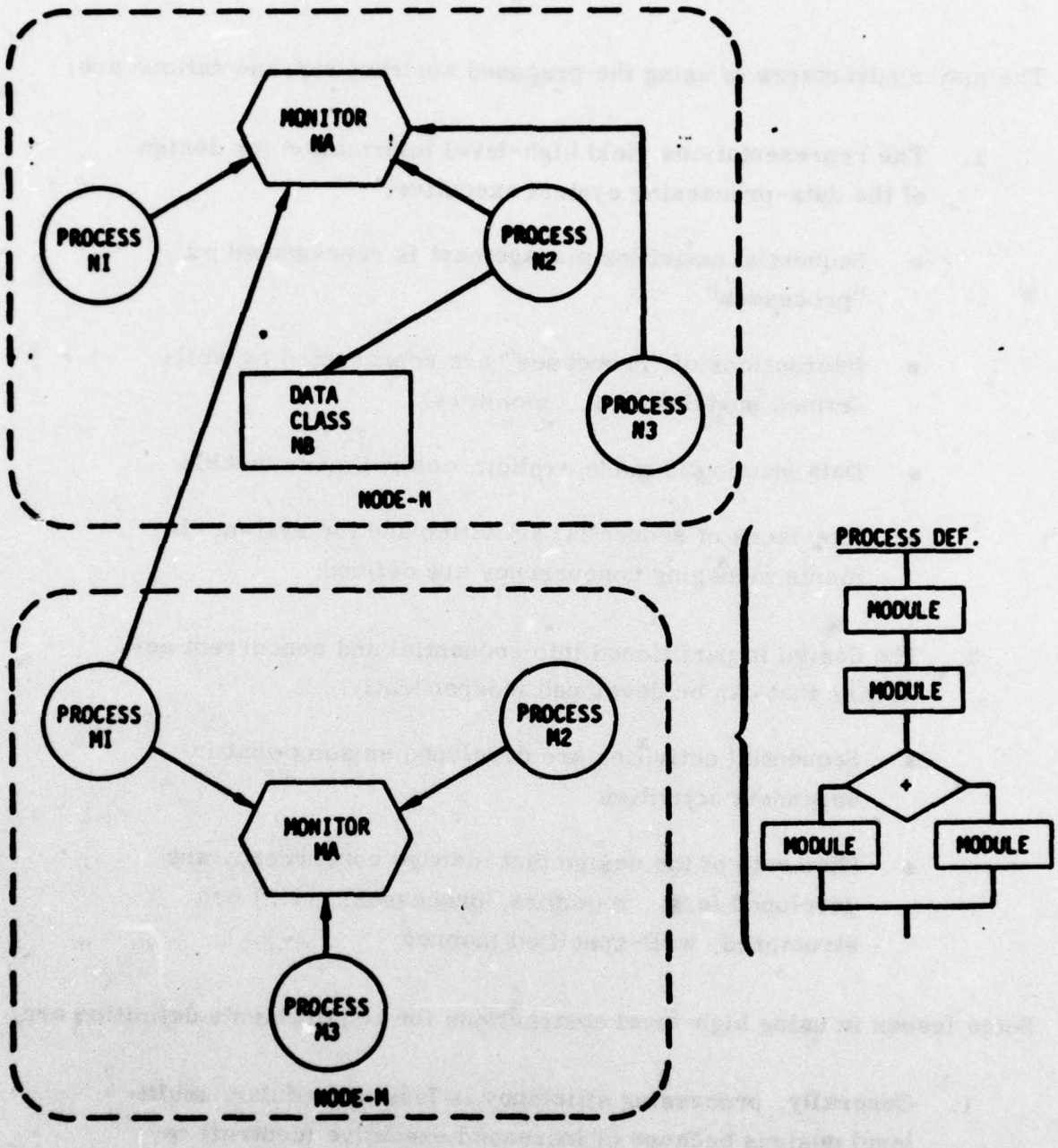


Figure 7. DDP Design Engineering Output Document Overview: Functional Requirements Portion

The major advantages of using the proposed abstract representations are:

1. The representations yield high-level information for design of the data-processing system executive:
   - Sequential activities management is represented by "processes"
   - Interactions of "processes" are represented by well-defined modules (e.g., monitors)
   - Data sharing is made explicit, control more visable
   - Interfaces of sequential activities and the system elements managing concurrency are defined.
2. The design is partitioned into sequential and concurrent activities that can be developed independently:
   - Sequential activities are developed as somewhat independent activities
   - Elements of the design that manage concurrency are developed (e.g., monitors, processes, . . .) in a structured, well-specified manner

Some issues in using high-level abstractions for requirements definition are:

1. Generally, processing efficiency is less in modular, multi-level designs because of increased executive (control) requirements.
2. Some concepts have yet to be dealt with:
   - Dynamic creation of processes

- Preemption of an executing process
- Priority of processes seeking the same resource.

3. The design approach is still experimental and therefore some risk exists that it has critical faults and/or low-performance situations.

Textual material also must be contained in the output document. The type of material is the same as described for the input document (Subsection 2.2), but at a different level of detail. (In particular, many of the unstructured requirements should have been transformed into structured requirement details.) The resulting overall output document contents and format are defined in Tables 7 and 8.

TABLE 7. DDP SUBSYSTEM DESIGN OUTPUT DOCUMENT CONTENTS

1. DP "Subsystem" Definition

   Substructure Identification

   - Components/Partitions (HW/SW processes, modules; data abstractions)
     - Functions, subfunctions ident.
     - Locations (at execution)
     - Data distribution
   - Connecting structures/interfaces (networks)
     - Data, control, test paths
     - Com. modes, media, rates
     - Executive support requirements

   Substructure Design Requirements

   - Data abstraction descriptions
   - Functional descriptions and operating rules (what, when, how)
   - Performance (quantitative)
   - DP architecture attributes

2. DP "Subsystem"-Level Requirements

   - Allocated DP subsystem requirements
   - Design criteria (payoffs)
   - Design constraints
   - Environmental factors

3. DP "Subsystem"-Level Testing Requirements

   - Plans, criteria
   - Simulations, support requirements
   - Mission and system requirements traces

TABLE 8. DDP DESIGN ENGINEERING OUTPUT DOCUMENT FORMAT

Graphical (Or Concurrent Language) Representations

- Identify data-processing nodes (platforms)
- Identify function abstractions (processes)
  - All processes are potentially concurrent
  - All processes are individually testable, traceable
- Identify data abstractions
  - Monitors and classes define process coupling via data access rights
- Identify subfunction abstractions (modules)
  - Processes are composed of modules
  - Modules are the smallest functional building blocks
  - Modules are individually testable, traceable
- Identify module coupling
  - No common data environments

Text

- Same considerations as for input document

SECTION 3
BASELINE DESIGN THEORY DEFINITION

3.1 DESIGN ENGINEERING MODELING

By viewing boundary conditions, we have obtained a "black box" concept of design engineering. To make further gains, we must assume some type of internal structure to the box.

In a broad sense, design is envisioned to be hierarchically (or top-down) structured. That is, design is pictured as consisting of a sequence of activities which progressively decompose and refine groups of requirements. Figure 8 illustrates a hierarchical design structure and indicates desirable characteristics of the design process and products.

Design of complex systems seldom can be structured like the simple hierarchy indicated because of the many dimensioned relationships of the many dimensioned relationships of the many design decisions. Consequently, for BMD DDP, we assume that much design iteration will be necessary to evaluate tradeoffs and understand full implications of decisions, but that an overall top-down progression will still be evident.

The overall top-down progression of BMD DDP design will be driven by the sequence in which requirements are accommodated by design decisions. Each requirement constrains the design, sometimes in conflict with other requirements. Efficient design techniques consequently must address requirements in decreasing order of "force" (or extent) of constraining effects; i.e., the most constraining and "forceful" requirements must be considered first. Otherwise, many candidate designs that would later be rejected may be unnecessarily examined during early design. (At the same time, care is needed to assure that potentially feasible solutions are not

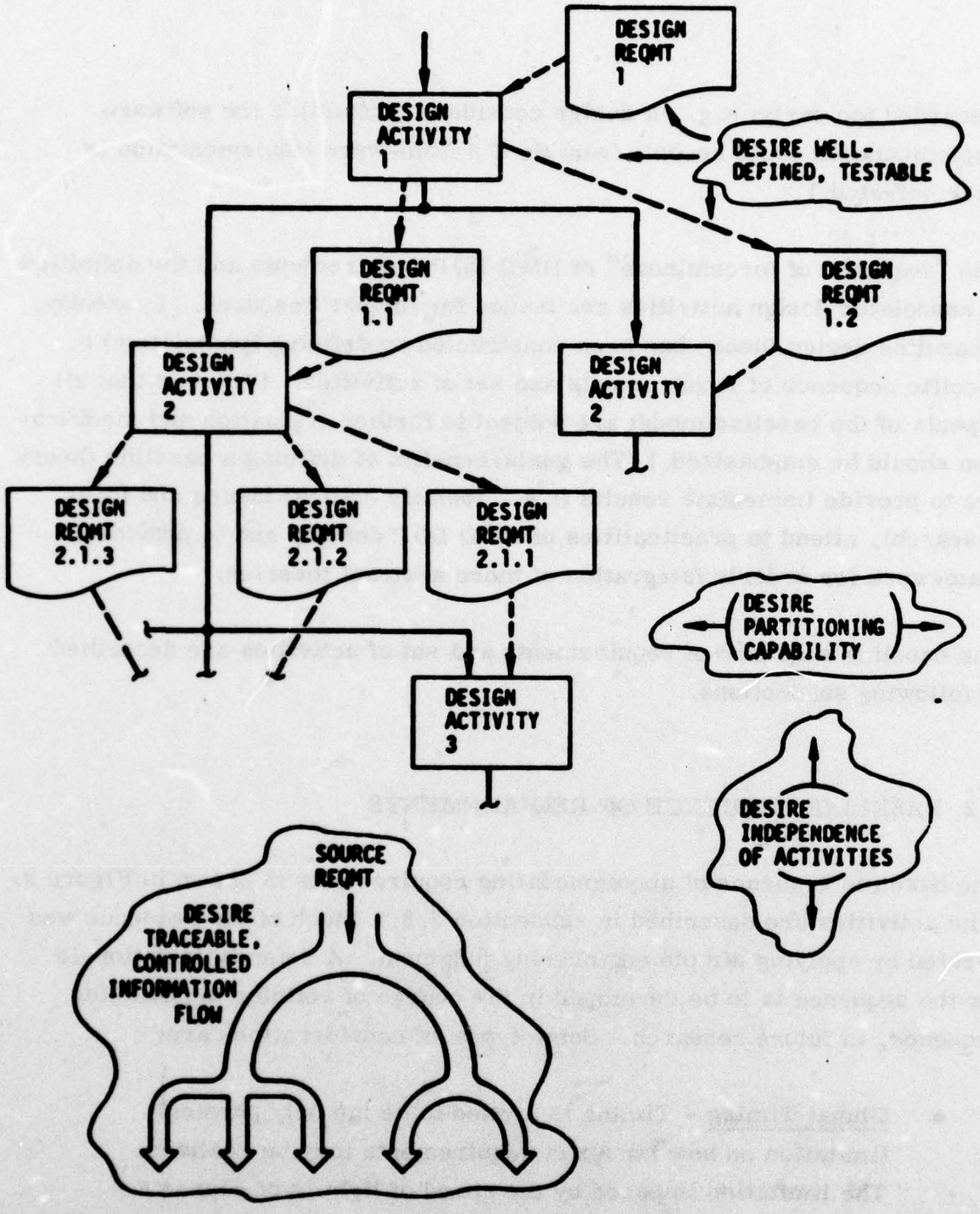


Figure 8. Hierarchical Design Structure and Desired Characteristics

discarded too early; e.g., a design considered infeasible for software implementation could become feasible if a firm-ware implementation is later selected.)

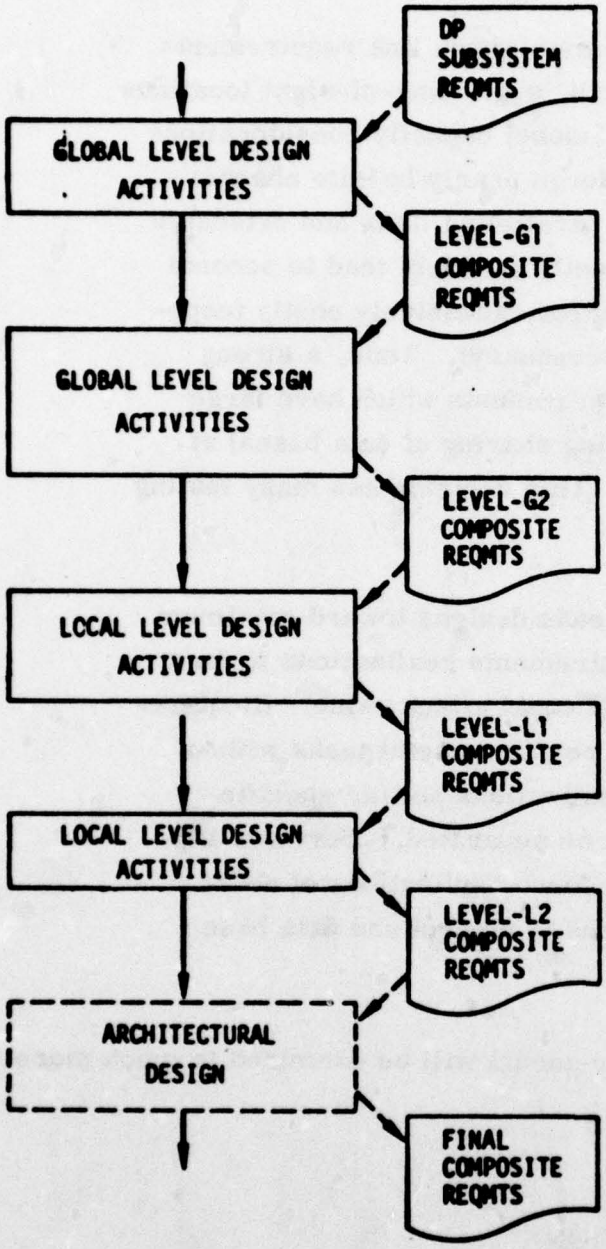
The "sequence of forcefulness" of BMD DDP requirements and the definition of associated design activities are issues for further research. Presently, a baseline design theory has been constructed by defining (postulating) a specific sequence of requirements and set of activities. (The fact that all aspects of the baseline model are subject to further evaluation and modification should be emphasized.) The goals/benefits of defining a baseline theory are to provide immediate results (i.e., identify critical issues and focus research), attend to practicalities of BMD DDP design, and to provide a framework for orderly integration of more abstract theories.

The baseline sequence of requirements and set of activities are described in following subsections.

3.2 BASELINE SEQUENCE OF REQUIREMENTS

The baseline sequence of accommodating requirements is shown in Figure 9. (The activities are described in subsection 3.3.) Much of the sequence was derived by applying simple engineering judgment. A defendable rationale for the sequence is to be developed in the course of refining the baseline sequence, in future research. Some types of considerations are:

- Global Timing - Timing is argued to be the only physical limitation on how far apart requirements may be realized. The limitation imposed by the speed of light is of course a hard constraint.



- STRUCTURED APPLICATION REQUIREMENTS
- CONSTRAINTS/RATIONALE/ENVIRONMENT FACTORS (UNSTRUCTURED REQUIREMENTS)

1 GLOBAL CONSTRAINTS IDENTIFIED
1.1 SPECIFIED GEOGRAPHIC CONSTRAINTS
1.2 ALL ENVIRONMENTAL CONSTRAINTS
1.3 GLOBAL TIMING
1.4 DISTRIBUTION OF CONTROL, DATA BASE
1.5 COMMUNICATIONS

2 GLOBAL REQUIREMENTS ADDRESSED
2.1 SURVIVABILITY
2.2 GROWTH (GLOBAL ASPECTS)
2.3 DEPLOYABILITY (GLOBAL ASPECTS)
2.4 LIFE CYCLE COSTS (GLOBAL ASPECTS)

3 LOCAL CONSTRAINTS IDENTIFIED
3.1 DISTRIBUTION OF CONTROL, DATA BASE
3.2 COMMUNICATIONS
3.3 THRUPUT

4 LOCAL REQUIREMENTS ADDRESSED
4.1 GROWTH (LOCAL ASPECTS)
4.2 DEPLOYABILITY (LOCAL ASPECTS)
4.3 GRACEFUL DEGRADATION
4.4 AVAILABILITY/RELIABILITY
4.5 LIFE CYCLE COSTS (LOCAL ASPECTS)

5 ARCHITECTURES DEFINED:
5.1 AVAILABILITY/RELIABILITY
5.2 THRUPUT
5.3 REAL TIME RESPONSE
5.4 DEPLOYABILITY
5.5 GROWTH (MODULE LEVEL ASPECTS)
5.6 LIFE CYCLE COSTS
5.7 HARDWARE UTILIZATION

Figure 9. Preliminary Sequence of Addressing Requirements in Gaseline Design Theory

- Global Communications - Communication link requirements impose certain hard constraints, e.g., line-of-sight locations for μ wave and laser links. Channel capacity considerations form softer constraints. Although nearly infinite channel capacity can be purchased by paralleling links and extensive message encoding, large-capacity channels tend to become unmanageable, unreliable, and/or excessively costly (especially when growth becomes necessary). Thus, a strong desire exists for grouping requirements which have large inter-communications (including sharing of data bases) at common geographical nodes. This decision has many testing implications, also.

- Survivability - Survivability leads designs toward maximum geographic dispersion of requirements realizations to form nodes of small, approximately equal attack value. (Requirement realizations that cannot perform useful tasks without other specific requirement realizations and/or specific external subsystems need not be separated.) Survivability requirements are expected to force replications of nodal requirements and complications in control and data base design.

The sequence of accommodating requirements will be examined in much more detail in future research.

3.3 BASELINE ACTIVITIES DEFINITION

The design model of subsection 3.2, Figure 9 was used to indicate the sequence of levels* of design. A more detailed model, Figure 10, is used in this section to help examine the design activities at each level.

*The level is determined by the requirement accommodation sequence.

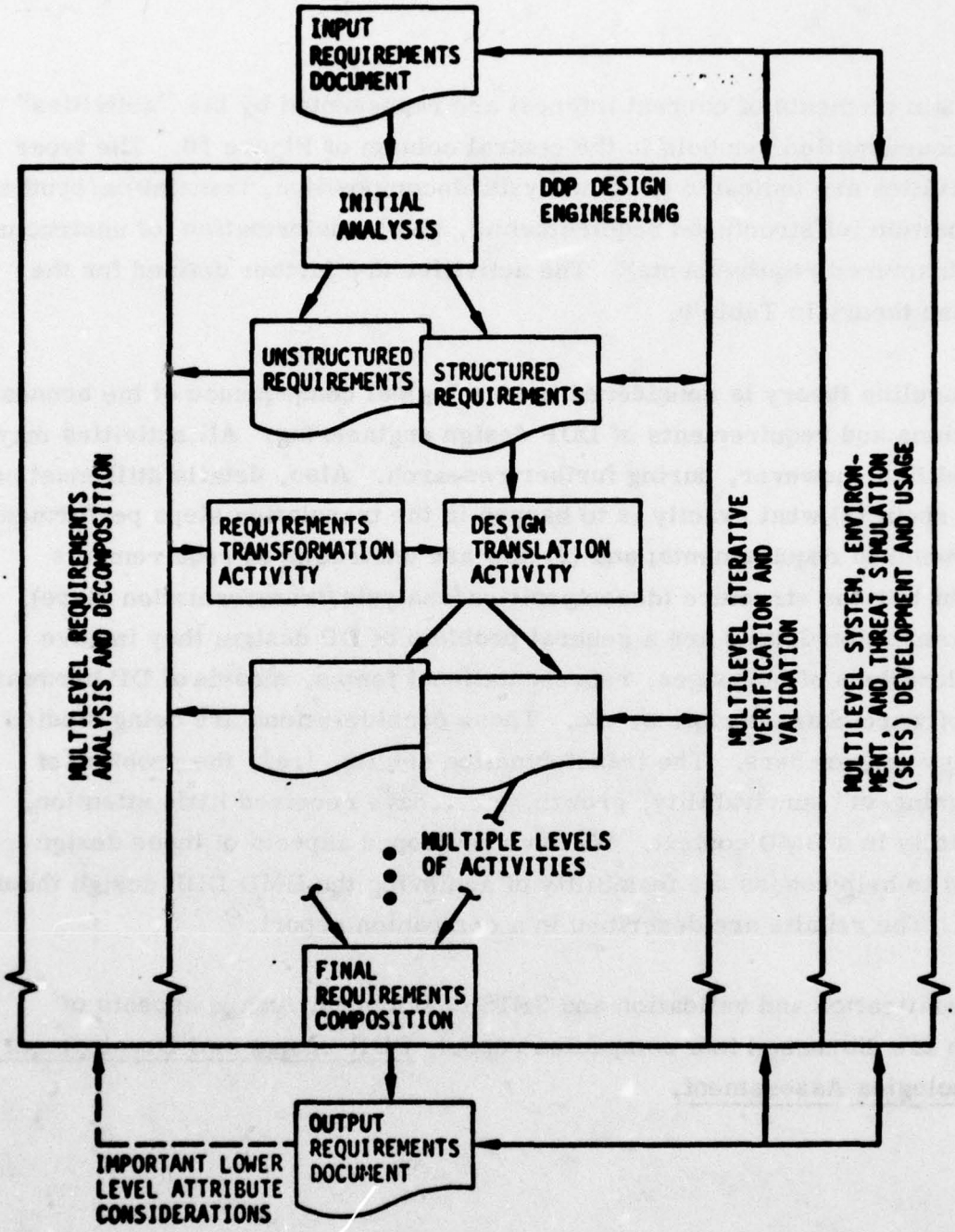


Figure 10. DDP Design Engineering Model Adopted as a Research Framework

The main elements of current interest are represented by the "activities" and documentation symbols in the central column of Figure 10. The types of activities are indicated to be analysis/decomposition, translation/synthesis/composition (of structured requirements), and transformation (of unstructured into structured requirements). The activities are further defined for the baseline theory in Table 9.

The baseline theory is considered to be a logical consequence of the boundary conditions and requirements of DDP design engineering. All activities may be modified, however, during further research. Also, details still must be added about (1) what exactly is to happen in the translation steps performed on structured requirements; and (2) how are unstructured requirements brought into the structure (decomposition/analysis/transformation steps). The translation details are a general problem of DP design; they involve considerations of languages, representational forms, models of DP hardware and software characteristics, etc. These considerations are being studied by many researchers. The transformation details, i.e., the problem of "designing-in" survivability, growth, etc., have received little attention, especially in a BMD context. We have developed aspects of these design details to help assess the feasibility of achieving the BMD DDP design theory goals. The results are described in a companion report.[2]

The verification and validation and SETS development/usage aspects of design are discussed in a companion report, DDP Design and Development Technologies Assessment.

TABLE 9. BASELINE DESIGN THEORY ACTIVITIES

1. Initial Analysis
   - Associate modules with applications subfunctions
   - Associate data abstractions with applications data entities
   - Verify module testing integrity; return to higher level
2. Multilevel Requirements Analysis and Decomposition
   - Analyze/decompose unstructured requirements into components for lower level design
   - Identify levels at which components are to be addressed
3. Global Level Design Activities
   - 3.1 Translate Requirements to Initial Process Architecture
     - Evaluate module coupling strength
     - Assign modules to processes
     - Assign local data abstractions to processes
     - Allocate performance to processes
     - Define process test and evaluation requirements
     - Test and validate
   - 3.2 Transform Applicable Unstructured Requirement Components into Structured Requirements at Global Level
     - Modify process structure
     - Constrain process locations
     - Add "induced" requirements (processes/modules and data abstractions)
     - Iterate with activities 2 and 3.1 as required
4. Local Level Design Activities
   - 4.1 Translate Requirements into Initial Nodal Architecture
     - Assign processes to nodes
     - Define global network
     - Add "induced" requirements for network communications
     - Iterate with activity 3 as required
   - 4.2 Transform Applicable Unstructured Requirement Components into Structured Requirements at Local Level
     - Modify nodal processing structures
     - Add "induced" requirements (processes/modules and data abstractions)
     - Iterate as required
5. Final Requirements Composition
   - Acquire module algorithms where specified
   - Assess module and process hardware and executive requirements
   - Define architecture - Determining attributes
     - Processor/special Hw types and capacities
     - Memory types and capacities
     - Executive characteristics

SECTION 4

DESIGN TECHNOLOGY RESEARCH OVERVIEW AND EXPECTATIONS

Preceding sections concentrate on high-level, procedural aspects of design, i.e., basic questions of "what-to-do." Most DDP design technology issues and research requirements involve further details of what-to-do and, additionally, questions of "how-to-do-it." In overview, the issues are tied to determination of the following aspects:

- How to accommodate each requirement in the design
- "Best" sequence of requirements accommodation (defines major levels of design)
- How to know when design at each level has been completed and/or when iteration is required
- How to evaluate design successfulness (performance)
- Details of what to do in each activity at each level (defines levels of abstraction at each major level of design)
- "Best" sequence of activities at each level
- Details of what are the inputs/outputs of each activity
- Graphical/language (representation) forms
- Verification, validation, testing, traceability requirements
- Simulations (of design products and SETS) construction/usage
- How and when to communicate design results to the customer and system development management (documentation hierarchy).

Complete resolution of the issues would constitute definition of a true DDP design theory, which would enable achievement of a high degree of design automation. The development of design theories for large, complex systems has received much attention in recent years for general system contexts [3, 4, 5, 6] and for general software data processing systems [7, 8]; many other references exist. The manifest result of these activities is the finding that such design problems cannot be solved in an absolute sense and, therefore, that associated design theories cannot achieve completeness nor be significantly automatable. We are dealing with a so-called "wicked" problem [4, 9, 10,] which is partially characterized by (1) inability to define the problem completely; (2) inability to state the problem without at least partly stating (biasing) the solution; (3) no stopping rule; (4) no absolute measures of correctness or falseness; and (5) no absolute list of admissible procedures and operations. In effect, existing results indicate that design must remain largely a creative activity, having a high degree of iteration and without a simply related, uniformly evolving set of products.

The impossibility of a fully defined and automatable design theory does not reduce the need for a design theory. It does mean that for any useful progress, early research in design theory must focus on important but tractable elements of the problem. We thus plan to concentrate on the what-to-do details rather than the how-to-do-it aspects of design. In effect, we are planning research into development of a design technology, rather than of a design theory. Of course, the possibility of advancing the theory may be improved during our nearer term research, and we may then be able to rationalize a shift of emphasis toward more formalism later.

In terms of the baseline theory, we are particularly concerned with technologies to support interfaces between activities at each level and between levels. These interfaces are points of transition from one creative activity to another, where automated aids can be employed to facilitate efficient achievement of complete, consistent, and traceable designs. Good interface definition will make the creative tasks simpler, less error prone, and more visable.

The fundamental role of the baseline theory (although *not* a true design theory) is seen to be to provide a framework for focusing research (at interfaces). It also ties together research to enable demonstrations of results in meaningful experiments.

With reference to the baseline design theory (see Figure 10 and Table 9), research is required to provide a supporting technology for interfacing the basic activities:

1.) Translation of structured requirements into processing architectures at each level of abstraction and each major level of design

2.) Analysis/decomposition of unstructured requirements into components

3.) Transformation of unstructured requirement components into structured requirements

The technology issues and research requirements associated with these three design activities are further illuminated through discussion of a generic system design problem.

Figure 11 illustrates a generic system design activity. The design activity at each level has been decomposed into eight basic elements. (The elements are generically the same at each level, but the level of detail, design primitives, tradeoff considerations, etc., represented by the elements are different at different levels.)

The research areas necessary to support the generic design activities of Figure 11 are as follows:*

1) Definition of the problem definition space
2) Definition of the problem technology space
3) Definition of the feasible solution space
4) Definition of solution evaluation criteria
5) Methodologies for searching for good solutions according to the criteria and evolving lower level problem/technology definitions and solutions
6) Means of demonstrating the adequacy and goodness of a solution
7) Mechanics of communicating and extracting information/direction between members of the design team, the customer, and the design aid computer
8) Development of computer support

Each of the research areas has activities which require long-term research. Although the research areas are treated separately here, considerable interconnection exists and no area can be developed in isolation of the others.

The three basic design activities--translation, analysis/decomposition, and transformation--are represented by element 5 in Figure 11. The remaining elements comprise the supporting technology, for which we are planning research. The nearer-term research is anticipated to focus on definition and use of the "spaces" (elements 1-4). Longer-term research would formally be in elements 6-8, although we consider these research areas to

*The terms, "space," "dimensions," etc., used herein are intended to depict general concepts without specific mathematical definition.

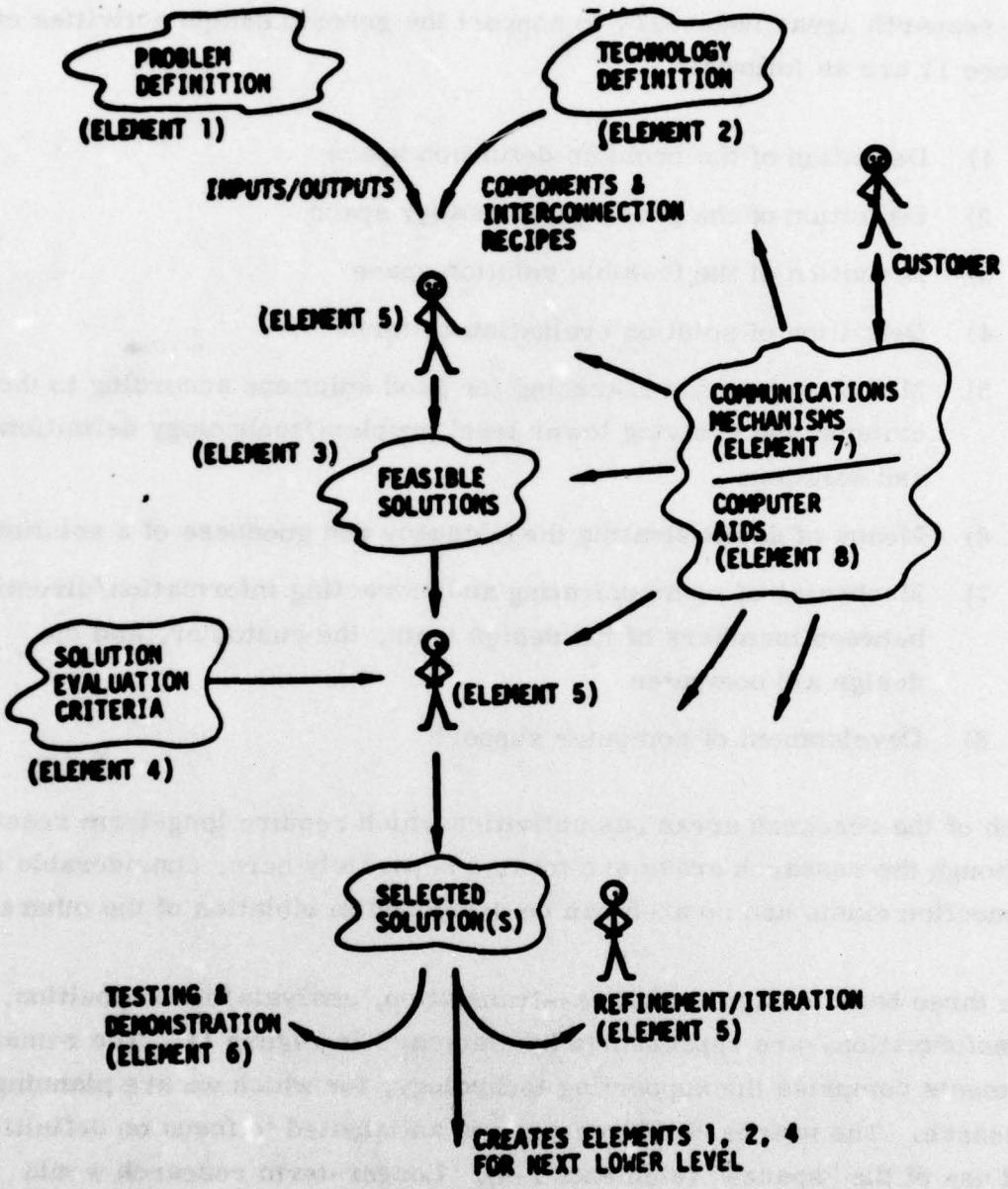


Figure 11. Elements of Generic Design Activity

be very closely coupled with all the other areas. The research activities are outlined further below.

4.1 PROBLEM DEFINITION SPACE

The problem definition space for a general system has been characterized by Wymore[5] as an input/output specification, defining all possible inputs and matching sets of outputs; the input/output set is often infinite. (We assume that environmental factors, etc., are included.) Certainly the assumption of such a complete specification would aid theoretical studies, but it is rather unrealistic.

At high levels, BMD DDP problems generally will have little more input/output specification than overall threat parameters, scenario outlines, and the allowable loss of defended value, if present specification procedures continue. Thus, many "textbook" design techniques have limited applicability; this is a major problem in trying to develop a design theory.

Research is needed to determine what input/output relationships and primitive descriptors are necessary and useful at various levels of design. Abstract representations of functionality, control distribution, and data base distribution requirements are minimally necessary. The representation of unstructured requirements is a major issue, and is closely coupled to the analysis/decomposition and transformation activities.

4.2 PROBLEM TECHNOLOGY SPACE

The problem technology space is characterized by Wymore as a set of descriptions of all systems which are buildable in the technology. (We assume that the technology descriptions correspond to best estimates for the time frame of interest.) The space must contain descriptions of

components available in the technology and "recipes" for their interconnection (including executive [operating system/data management] functions for the case of DDP design).

The problem technology space will have many dimensions for general DDP systems because of the large numbers of components and interconnections that are conceivable. Existing interconnection taxonomies are a start. But the real problem appears to lie in integration of types of components, communications, and executives with the interconnection structures to form a complete, overall taxonomy.

The coordinates of the space must have some relationship to those of the problem definition space so that a feasible solution space can be constructed. The coordinates must also relate to the solution evaluation criteria so that good, feasible solutions can be found. Wymore specifies the following components for a system description: name; time scale (response time indication; descriptions of all possible states, all possible inputs, state transition functions, and outputs while in each state; and various models that provide evaluation data (e.g., reliability). Use of this textbook characterization appears unrealistic in the BMD DDP context.

4.3 FEASIBLE SOLUTION SPACE

The feasible solution space is conceived to consist of points which correspond to all systems which both satisfy the problem definition specification and are buildable in the technology. It is the space defined by a joint mapping of points in the problem definition and technology spaces. (Usually the feasible solution space does not exist in reality, except as indicated by candidate designs.)

A point in the feasible solution space in effect corresponds to the assertion that a system (components and interconnection recipe) selected from the technology space will produce required outputs from associated inputs defined in the problem definition space. If the technology descriptions are complete, the mapping corresponds to a many-dimensional search of the technology space. Otherwise, the definition of valid assertions to provide such a mapping is expected to be extremely difficult. It corresponds to the basic design evolution procedure.

4.4 SOLUTION EVALUATION CRITERIA

Solution evaluation criteria for BMD DDP systems have been discussed often. They include survivability, reliability, etc. The major existing problem is the definition of valid, quantitative evaluation models for systems in the technology space for various subsets of points in the problem definition space. The models generally must exist at a number of levels of detail and be amenable to procedures for searching for "good" solutions.

4.5 DESIGN EVOLUTION PROCEDURE

Current design procedures for DP systems are said to contain steps of creation, and thus cannot have a complete theoretical basis. We must be satisfied with deriving guidelines for the procedures.

Some procedures involve where to start and what direction to take. The old "hardware first" procedure consisted of localized searches around points in the technology space to define a system which would satisfy the problem definition. The "software first" procedure attempts to construct a system in the technology space from a set of points in the problem definition space. Research areas here could include: (1) dual procedures whereby successive starts are simultaneously made from both spaces as necessary to efficiently

produce good or optimal designs; (2) automatable, adaptive solution search procedures; and (3) design catalogs which provide wide coverage of the feasibility space, so that design becomes a refinement of a cataloged structure.

Other procedures focus on detailing the sequence of steps to be taken. They include partitioning and decomposition techniques. Further research is needed on selection of consistent attributes for partitioning/decomposition and the sequencing of binding design decisions.

4.6 OTHER ELEMENTS

Testing and demonstration (element 6), communications mechanisms (element 7), and computer aids (element 8) depend highly on research in areas of the other elements. Much work has been done in these areas --including R-nets/RSL/REVS[12], PDL/PDE[13], Fitzwater's characterizations[14], and many others. Future research should consist of attempting to apply this existing formalism in the research areas described in preceding sections. This should make any shortcomings of the existing technology apparent and lead to necessary refinements.

More specific DDP design technology issues and research requirements are discussed in following sections.

SECTION 5
SPECIFIC RESEARCH REQUIREMENTS

5.1 DESIGN REQUIREMENTS TRANSLATION

Research is required to support translation of structured requirements through levels of abstraction. The near-term research should concentrate on the primitive entities and operations necessary and useful for describing requirements at each level of abstraction. Primitives must be identified to support both graphical and textual requirements representations. Based on these primitives, the long-term research should develop associated descriptive language(s) which can support automated processing and testing.

Issues to be resolved in the near-term research are: can the system requirements and the design decisions be described in terms of the specified primitives and in the format prescribed to delineate their interrelationships? Specifically, is the set of primitives and the format--

1) Complete. Can all the necessary system design requirements, parameters and decisions be expressed in these terms, both at the input to a design step and at the output?

2) Feasible. Is the codification of design in these terms possible with the knowledge/information available to the designers?

3) Convenient. Does the expression of the design in terms of these descriptors aid or hinder the speed or ease of the design process? To what degree?

4) Useful. Does the use of these descriptors yield a "better" design? In particular,

a) Is there more awareness of the tradeoffs between the various design requirements vectors?

b) Does the imposed structure help to specify how to achieve the unstructured requirements? Can the designer see more easily how to enhance the system capability in these areas?

c) Does the imposed structure enhance the visibility of the design features? Are the factors which affect the unstructured requirements more visible? Can the designer see more easily the degree to which the requirements are not satisfied?

d) Does the imposed structure help to enforce consistency of design? Does it reduce design variability?

Description language(s) (DL) must be defined prior to serious use of the DDP design technology. (It is not reasonable to have as design goals both the definition of the DL and the system being described by it.) Thus, longer-term research is necessary.

The DL must be based on a number of primitive concepts, and composition rules for those concepts. The primitive concepts should suppress details that are not relevant. For example, consider designing the ordering relationships between computations. Suppose that the design goal is to describe only those orderings that are necessary for correct operation. Primitive concepts that allow one to describe partial orderings are inadequate. When computations can proceed independently, it would be inappropriate to prescribe an arbitrary ordering. Thus, the DL must allow the specification of partial ordering relationships in which the sequencing details of independent calculations are left unspecified. For example, a Petri-Net does not prescribe the order of concurrent events.

The structuring concepts of the DL define how primitive can be composed to describe more complex concepts. The structuring rules should provide naming conventions that allow references to substructures of a complete description. These substructures are another means of coping with complexity. It is assumed that another goal of design is to partition a DPS description into a collection of one or more elements that can be further described with a minimum understanding of their context. Named substructures provide a convenient means of describing the components of a partition. The overall description will thus be structured and hopefully separated into elements that can be further designed with only the knowledge of their interfaces without knowledge of how the remainder of the system is designed.

It is assumed that the design theory may require several DLs in a sequence of steps. Each step will fully describe the design using the available set of primitive concepts. Each description will necessarily leave details to be resolved in order to manage the overall complexity of the system. Each step of the design can interpret the description of the system to discover or test the properties of the design. At the highest levels, interpretation is accomplished with simulation; at the lowest level, interpretation is accomplished by direct execution.

It is presumed that some design levels are delimited by the introduction of a new DL. The description at level i is expressed in terms of primitives and structures or compositions of the primitives. Two alternatives appear to exist for evolving the description of the DPS from one level (step) to the next. The alternatives are to define the primitives, leaving the structure intact, or to translate both the structure and primitives to a new DL, preserving the meaning of the original description.

The former approach is similar to the use of abstract machines. At level i the details of the machine operations (primitive) are not fully defined. At level i + 1, a new set of structures and primitives are introduced to describe the primitives of level i. Functionally, this might be represented as

$$\text{System description} = f(s_i, p_i) \text{ at level } i$$

$$f[s_i, g(s_{i+1}, p_{i+1})] \text{ at level } i + 1.$$

An example of this approach is interpretation.

The latter approach implies that the DL of level i + 1, is capable of expressing all of the concepts and structures used at level i, as well as possibly introducing more detailed concepts to the system description at level i + 1. The system description at level i must first be translated to an alternative description at level i + 1. The details are subsequently added to this description. Functionally, this might be represented as

$$\text{System description} = f(s_i, p_i) \text{ at level } i$$

$$g(s_{i+1}, p_{i+1}) \text{ at level } i + 1.$$

An example of this approach is compilation.

When the concepts of a description language are finalized, they define an interface between applications described using those concepts and an abstract machine that interprets the concepts, and thus hides detail from the user of the system. For distributed data processing, it is of particular interest to determine whether the abstract machine(s) should make the network, nodes, and/or processor structures transparent to the application. Such transparencies offer the potential for ease of growth of the application and simplifying the design by logically isolating difficult functions like

resource management, fault detection, isolation and recovery, etc. However, these benefits must be traded off against the performance degradation encountered in such a design. For example, an abstract machine that completely hides the structure of the distributed data processing system from the application may preclude the application taking advantage of special-purpose hardware that might otherwise have a significant impact on performance.

The features of the abstract machines must also consider the need to support the programming languages that might be employed in constructing the application. Such languages may reflect machine dependencies (e.g., PFOR, a programming language for a SIMD architecture) and may preclude making the network topology transparent to the user. The emphasis on flexible software growth and reliability has encouraged the investigation of languages supporting modular programming concepts (e.g., Simula, Concurrent Pascal, Euclid, Modula, and Model). Such languages require a significant review of the abstract machine architectures needed to support their features.

The abstract machine may be realized entirely in hardware or it may be realized using hardware, software and/or firmware. Generally, abstract machine concepts can be implemented with software, given any hardware, at the expense of performance. For the application of interest, high performance is a major goal and therefore suggests that hardware considerations surface early during the design process. This raises a question regarding the role of hardware considerations in defining the features of the abstract machine. Should the available hardware technology drive the features of the abstract machine or should the features of the abstract machine levy requirements on the hardware designer. Firmware offers an implementation medium that is considerably more efficient than software and yet more flexible than hardware for an abstract machine implementation. The extent of its use to replace hardware and software is an open issue of great potential benefit.

5.2 DESIGN REQUIREMENTS ANALYSIS/DECOMPOSITION

The analysis/decomposition technology supports conversion of (perhaps) broad, unstructured requirements into specific "requirements vectors," which can be transformed into structured requirements. The conversion is treated here in two steps: (1) analysis/decomposition of the requirement, itself, to provide specific, unambiguous requirement components (or subrequirements); and (2) construction of a vector of elemental design responses for each requirement component. (A requirement vector is now conceived to be a linear array of elemental design responses; possibly, higher dimensioned arrays may be found necessary.)

One approach to the requirement decomposition step is to use a "requirement decomposition tree," such as illustrated in Figure 12 for growth. The tree is used as a template for queries of the requirement source to assure an unambiguous requirement definition. For example, a certain growth (change) accommodation requirement may be traced through the structure and found to involve the O&M phase, at peacetime, all types of threat changes, and (certain specified) BMD system tactics changes. In addition the effects to be maximized or minimized and (hopefully) some type of "utility" value additionally would be defined for each requirement component. For this example, then, the growth requirement could be decomposed into components structure shown in Figure 13.

The second step generally would be carried out for each requirement component identified in the requirement decomposition step. (The level of decomposition used to identify components may vary for specific cases; e.g., in Figure 13, the components may be selected in the following ways: [Req. 1]; [Req. 1.1, Req. 1.2]; [Req. 1.1.1, Req. 1.1.2, Req. 1.1.3, Req. 1.2]; and, perhaps, as some redundant combinations.)

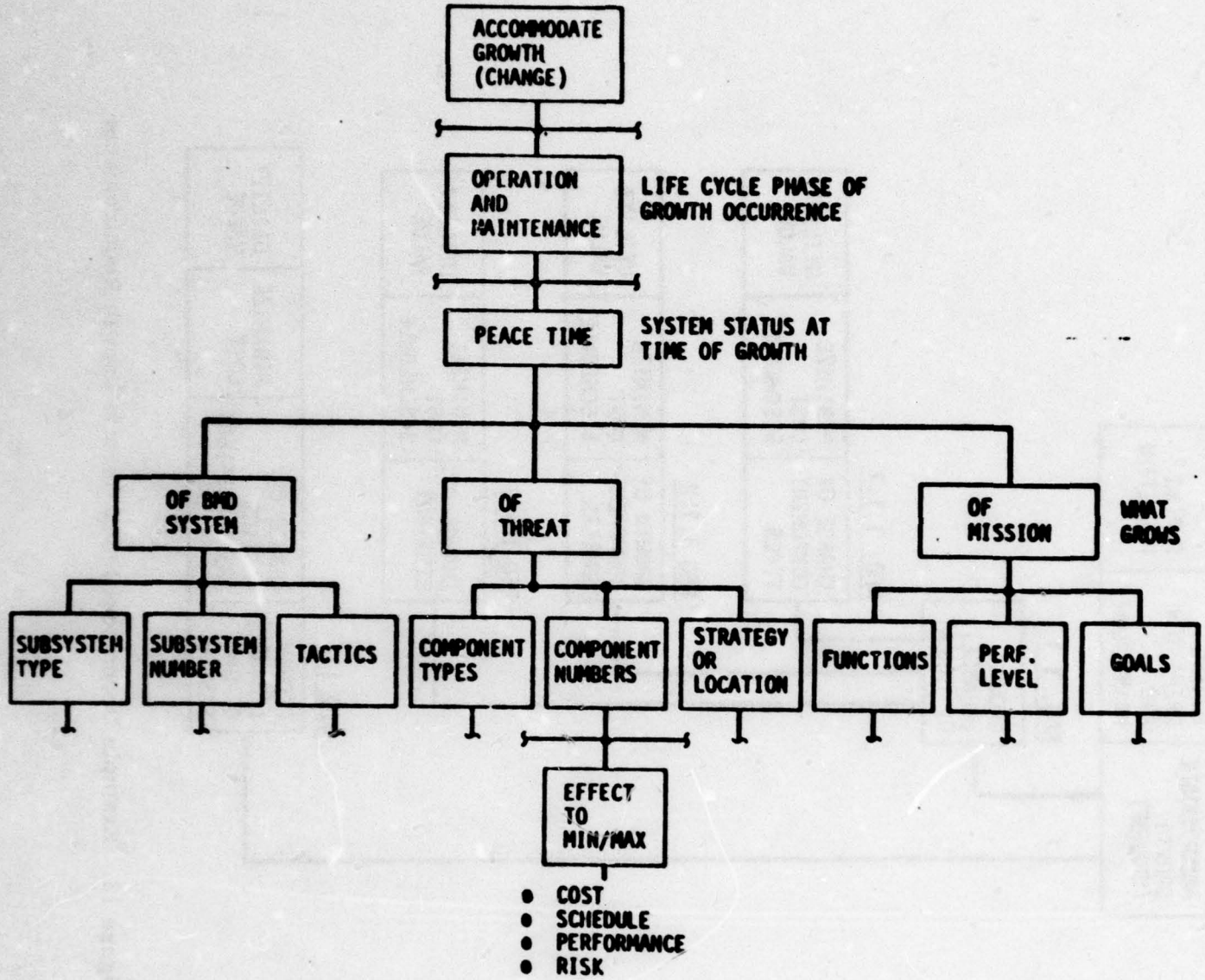


Figure 12. Example Requirement Definition Decomposition

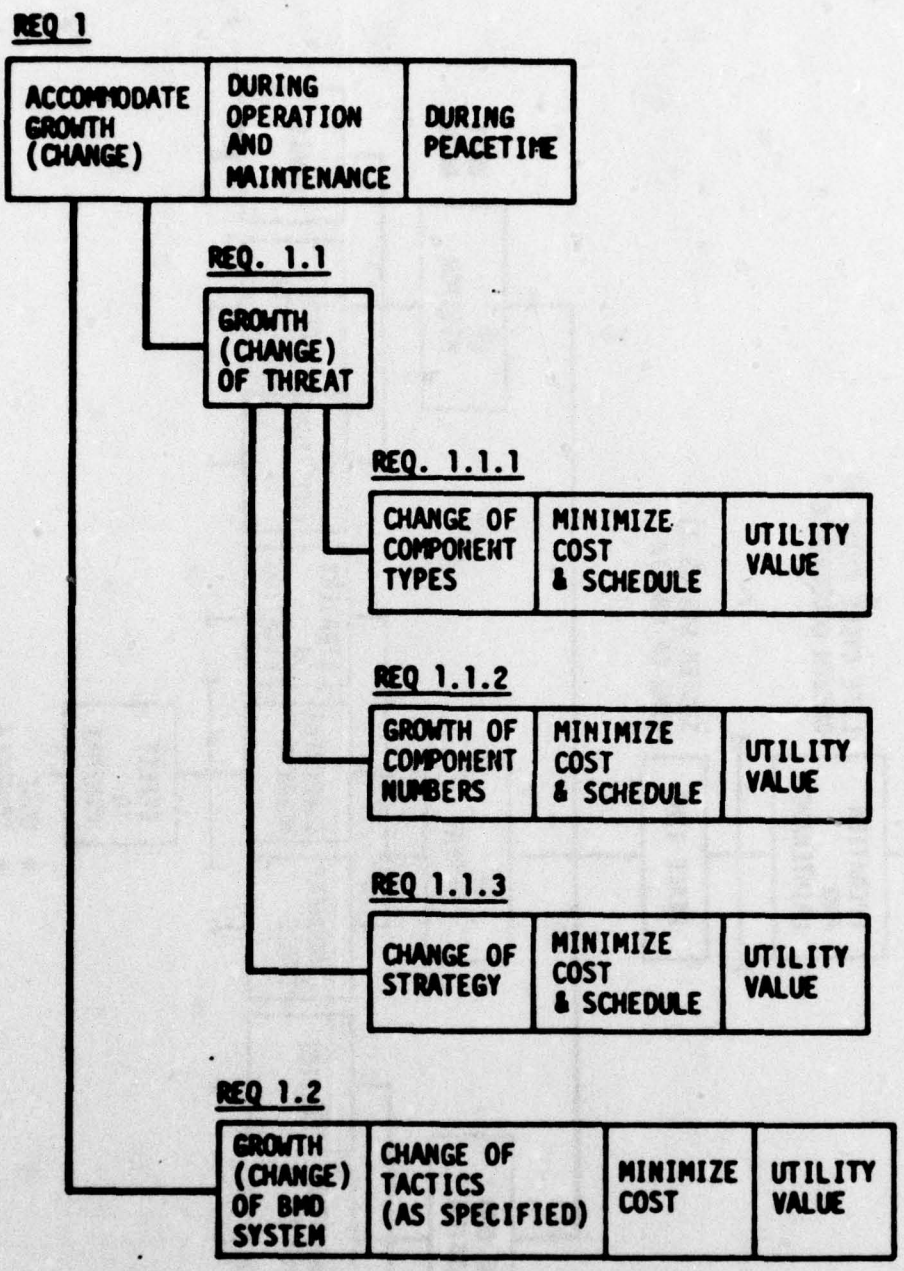


Figure 13. Example Decomposition of Certain Growth Requirements

A "response decomposition tree" as shown in Figure 14 may be used for the second step. In concept, a tree must be available for each possible requirement component; the component is used as a key to select the proper response tree. Figure 15 is an example response tree for Req. 1.1.2, growth of threat component numbers.

A requirement vector would be formed for the example of Figure 15 by collecting all of the leaves of the tree. For the present case, using identified leaves, the result would be:

- Requirement Vector: Growth of Threat Component Numbers
  - At "define process structure level," partition growth elements into separate modules
  - At "define network architecture level," construct an appropriate (e.g., star) network
  - At "select processors level," provide
    - -- Redundancy
    - -- Oversized CPUs
    - -- Oversized memories

The issues and research requirements for developing a requirements analysis/decomposition technology as described above are:

1. Selection of the most efficient analysis/decomposition structures; the tree structure illustrated is perhaps the simplest. Many other techniques exist for consolidating descriptive data, however. In particular, data base management studies should be applicable, especially concerning "query" languages being developed.[15]

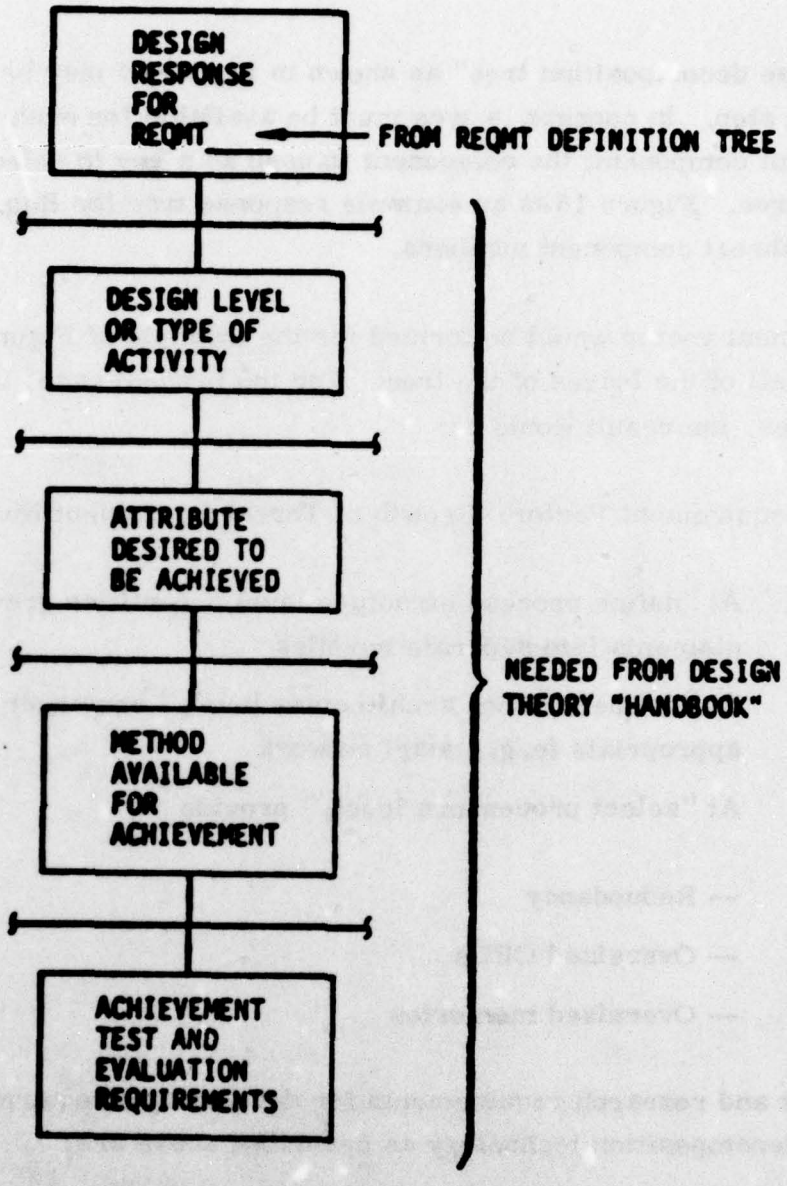


Figure 14. A Design Requirement Response Tree

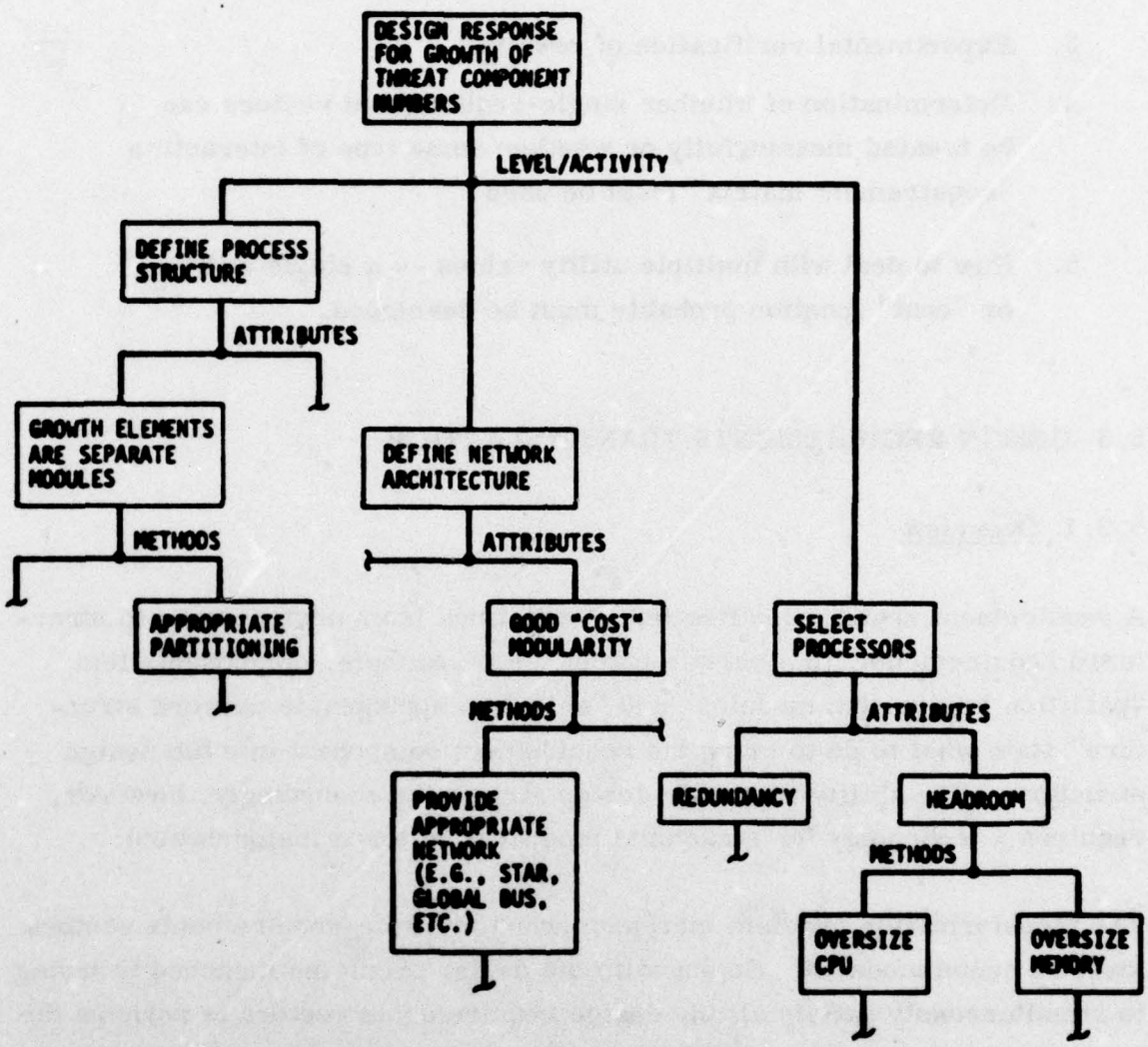


Figure 15. Example Response Tree for Growth of Threat Component Numbers

2. Construction of analysis/decomposition (query) structures for most typical BMD DDP requirements

3. Experimental verification of results

4. Determination of whether single-requirement vectors can be treated meaningfully or whether some type of interactive "requirement matrix" must be used

5. How to deal with multiple utility values -- a single utility or "cost" function probably must be developed.

5.3 DESIGN REQUIREMENTS TRANSFORMATION

5.3.1 Overview

A requirement vector specifies transformations from unstructured to structured requirements, to a certain extent. For example, components like "partition into growth modules" and "select an appropriate network structure" state what to do to bring the requirement component into the design structure. The ability to modify design structures accordingly, however, requires a technology for structural modification (or transformation).

The transformation problem increases when multiple-requirements vectors must be accommodated. Coping with the design problems incurred in trying to simultaneously satisfy all the design requirements vectors is perhaps the most challenging problem which exists for a DDP designer. The specified requirements tend often to be in opposition to one another, rather than supportive, which implies that excess capability must be originally designed into the system to compensate for any cancellation imposed by conflicting requirements. For example, survivability may require physical (geographical) distribution which implies greater communications and control overhead which tends to reduce performance.

We have identified three basic approaches or conceptual tools to provide design capability to deal with the requirements vectors:

- Control Formalism - The description of control requirements by means of a set of abstractions and a format for their specification which clearly delineates the interactions between system components. This technique should enhance visibility of tradeoffs in the control area.

- Modularism and Distribution of Capability - The identification of "natural" groupings and partitions of functions and data abstractions to reduce system complexity. Proper module selection yields minimal intermodule interactions.

- Parallelism - The arrangement of systems tasks so that they can operate concurrently. This is the basic tool to enhance both computational speed and system reliability. Parallelism can be established between both similar and dissimilar tasks; it can also be viewed as both a design philosophy and as an implementation tool.

These design approaches, all of which can be used simultaneously, provide a means of identifying, and to some degree, isolating the interactions of the various requirements and thereby provide a basis for tradeoff analysis. Therefore, they can aid a designer in choosing between possible alternatives which he has identified. For example, modularity establishes boundaries on which physical distribution is appropriate. Parallel implementation of those modules can provide enhanced performance and/or increases in both reliability and survivability. Control formalism specifies the required intermodule communications and makes clear the tradeoff between efficient, low-overhead implementation and more structured (monitor-like) implementation which provides better fault tolerance and V&V. Since thin-wire communication (which may conflict with performance) may be required if

the modules are distributed, there is strong interaction with the requirements for survivability and reliability.

In the following subsections, these design approaches are examined in more detail to illustrate their usefulness in the task of designing systems which simultaneously satisfy the set of design requirements vectors.

5.3.2 Parallelism

Parallelism is a valuable design tool when applied to either a set of similar (or same) tasks or dissimilar tasks. In the former case, parallelism can serve two functions. If the input data stream has multiple elements, all of which require similar processing, then a vector processor (parallel array of computing elements) can provide greatly reduced system computational time. If the input data stream consists of single elements, then parallelism is chiefly added to increase reliability through a redundancy. In the case of dissimilar tasks, the partition of the set into independent streams of tasks to be run in parallel can reduce computational time, provided that the necessary coordination can be accomplished. We will divide our discussion of parallelism along these lines, looking first at its application to same or similar tasks.

Parallelism Applied to Same or Similar Tasks -- The most basic technique of reliability theory has been the (parallel) replication of switches, gates, devices, etc., all assumed to have independent, identical failure characteristics. Such analysis has typically assumed that the same algorithm is implemented, with the parallelism realized by theoretically identical devices. However, this notion can be extended to a higher level to consider the parallel implementation of different algorithms for the same (or similar) task. Considered at this level, parallelism provides tolerance for algorithmic faults, and allows for the selection of best algorithm according to some

criterion. Such algorithmic parallelism is particularly useful when it is difficult or impossible to decide which algorithm is better based on the inputs (a priori), but is is possible to make such a choice given the outputs (a posteriori). An example is given below from the world of digital processing of speech signals, in which the accuracy of some of the algorithms is strongly data dependent. The linear prediction algorithm is generally best, but is occasionally subject to wild variations, which are detectable by examination of the residual estimate from the predictor. At such a time a less accurate, but more stable (and therefore reliable), estimator such as that given by cepstral analysis should be used. Although it is sometimes possible to predict when the linear prediction algorithm will fail from examination of the input data stream, the residual output is a much more reliable indicator. Thus the parallel implementation of these algorithms, together with a decision mechanism to decide which is best, will provide a consistently more accurate estimator of the speech characteristics than either algorithm alone. Thus parallelism has been used in this case to provide fault tolerance at the algorithmic level.

Algorithmic parallelism can sometimes also be used to improve performance. As an example, consider a system where a number of iterative algorithms are implemented in parallel to perform the same computation, and the result taken from the first algorithm to converge. In this case, "best" is measured in time to completion, and the effect on the system is reduced computation time.

The concept of algorithmic parallelism can be of benefit to a designer both as a design philosophy and as an implementation tool. When parallel algorithms are implemented directly, it implies a trade of additional hardware (computational capability) for algorithmic fault tolerance. Such implementation may be amenable to specialized HW/FW configurations for a particular task. As an example, consider the speech signal processing system shown in Figure 16.

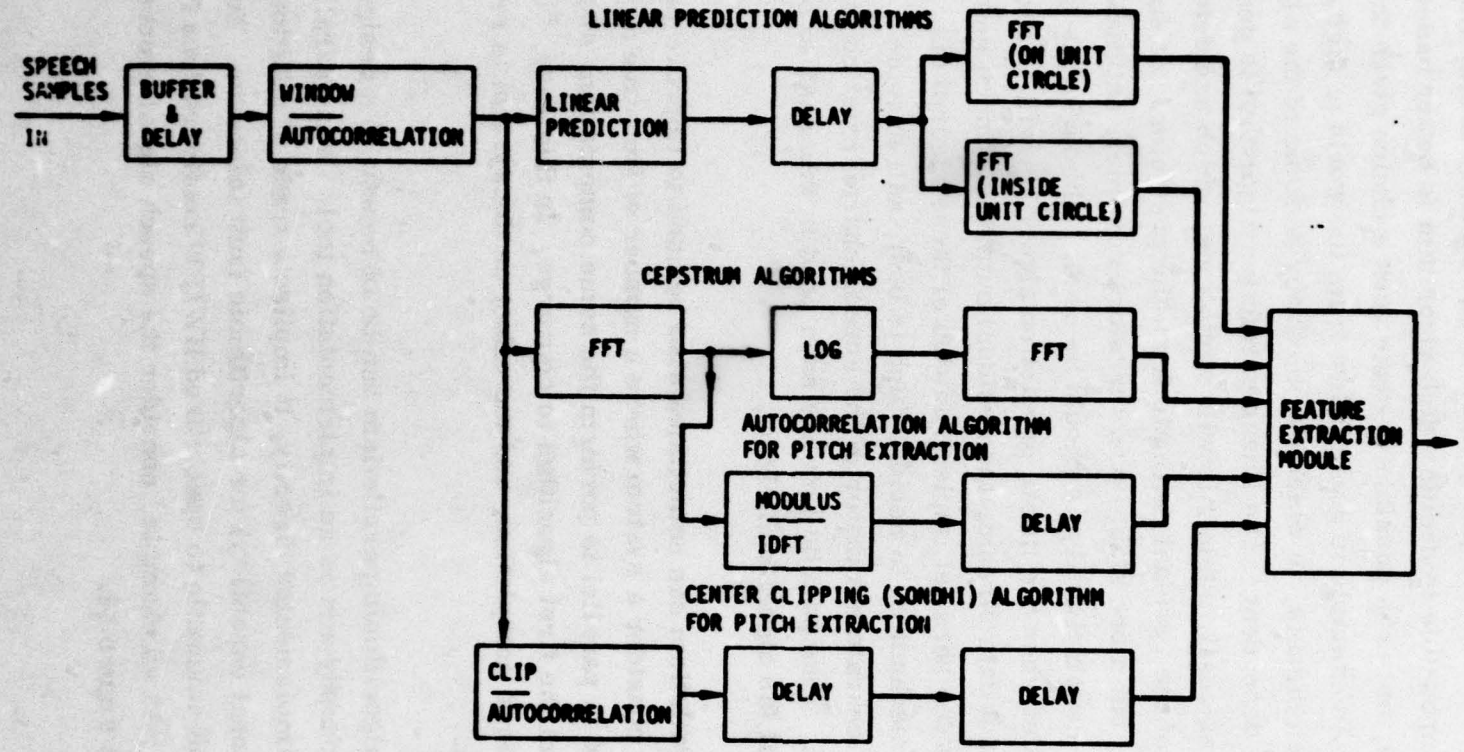


Figure 16. A Parallel Pipeline Architecture for Real-Time Speech Feature Extraction

This figure shows a parallel pipeline architecture designed for real-time digital signal processing. It accepts speech samples from an A/D converter and outputs, 5 cycles delayed, a "best" set of features extracted from the speech data. With the exception of the delay blocks, which are simply memory modules with communication hardware, all system blocks are configured with identical micro-programmable microcomputers. This greatly simplifies maintenance and availability (one spare serves all units), reconfigurability, and initial construction. The current implementation uses AMD2901 4-bit slice microprocessors with 1K words of memory. The system operates as a synchronous pipeline, with all blocks transferring data at a given time and then resuming operation. This allows for a very simple control and communications process which is totally transparent to the module software. The algorithmic parallelism provided is apparent from the figure. The upper paths provide two versions of linear prediction, the next a cepstral approach, and the lower two provide two independent algorithms for pitch extraction. The accuracy and reliability of these algorithms varies, depending upon the nature of the input speech data stream. The feature extraction module is programmed to select/compute the best a posteriori estimate of the desired features based upon the outputs of all five algorithms.

Although this example is very specific to speech signal processing, similar problems exist in many applications of digital signal processing. The parallel pipeline architecture illustrated can provide a simple, yet extremely powerful, tool for the solution of these problems.

Algorithmic Parallelism as a Design Philosophy -- The familiar technique of brainstorming, in which as many solutions to a problem as possible are suggested, can be viewed as conceptually similar to algorithmic parallelism. The key feature of this approach is that no solution, no matter how implausible it seems, is rejected out of hand; solutions are discarded only when it has been shown conclusively that they are not feasible. (It is alleged that the use of an iceberg as a fresh water supply, now seriously proposed for

implementation in the Middle East, was first suggested in a brainstorming session at RAND many years ago.) In the DDP construct, this notion translates into dual concerns that all feasible algorithms be identified as early as possible, and that algorithms not be discarded too early in the design process. The principal cause for the latter concern is the effect of firmware implementation of algorithms on performance. Such special-purpose tailoring of the machine architecture may make feasible many algorithms which were previously infeasible within realistic real-time constraints (as was true in the speech signal processing example given earlier). One problem that is faced in DDP design is that the high-level designer may not fully understand what is feasible in special-purpose (firmware) implementation and may therefore exclude from consideration some feasible algorithms.

Another problem is that the designer must balance the desire to find the best system algorithm against the increase in complexity of the design process which results from consideration of many options early in the design process. Thus the designer is faced with the dilemma of wanting to expand his horizons on the one hand and narrow them on the other. His only hope is to consider first those requirements for which both the technology is well known, and the constraints imposed by the feasible solution set strongly restrict the design space. In this way, he can remove the clearly infeasible solutions early in the design process, while still maintaining design breadth through parallel consideration of multiple algorithms.

Algorithmic parallelism is also useful as a testing tool. For example, the designer can simulate various algorithms in parallel to decide which is best. In so doing, he can use a technique similar to the control variate approach[17] to reduce the simulation run length requirements, as follows: if the same random inputs are used in all the (parallel) algorithms and if "like" decisions, across all the algorithms, are decided by reference to the same random number, then the outputs of the algorithms should be the same or at least very highly correlated. This implies that the variance of the difference

between the results of two algorithms should be much less than the variance of either result itself. Hence, the algorithmic comparisons can be made with many fewer runs than would be the case if each simulation were conducted independently. Moreover, the parallelism also provides a partial validation of the simulations since the answers from the various algorithms should be the same or nearly so. Therefore any major deviations in the results would be indicative of simulation (or perhaps algorithmic) error. Clearly, this approach suffers from the common testing problem of checking consistency between solutions rather than correctness. It should also be emphasized that the criteria for the choice of the best algorithm may include questions of survivability, growth potential, V&V, etc.

Algorithmic Parallelism as an Implementation Tool -- If the choice is made to implement several algorithms for the same task, this implies that the software/firmware to perform those algorithms are truly independent since the detailed specifications will be different as will the coding. If these algorithms are then implemented on parallel hardware devices, fault tolerance is achieved at all levels: algorithm, software, firmware, and hardware. Thus fault tolerance is achieved even for intermittent errors, such as those which are data dependent or due to improper timing, whether they occur in the algorithm or in its (HW/FW/SW) implementation. Moreover, because of the independent implementation, any problems with incomplete, incorrect, or inconsistent specifications are more likely to be detected since the parallel algorithms provide a self-checking feature. Unfortunately, the error will not be identified or isolated, unless only one from a set differs in result from the rest, but the occurrence of different results from "similar" algorithms is sufficient to detect the presence of one or more errors. Similarly, this self-checking feature is also helpful as a tool for detecting errors of any sort in the SW/FW/HW implementation, and reduces the V&V burden on the designer. It may also reduce the requirements for the SETS module to generate correct outputs for a given exercise, since the self-checking of the algorithms may suffice for that purpose, perhaps through a majority rule or other voting algorithm.

Finally, parallel implementation of algorithms may prove advantageous for adaptive learning/evolutionary systems. If there is a question about the stability or implementation of the adaptation algorithm, a base-line system could be run in parallel to provide both a partial check and a back-up in case of failure. Similarly, in evolutionary systems, provision for parallel algorithm implementation would allow a copy of the previous (old) system to run concurrently with the updated system for some period in time, thus providing a continuous check of the reasonableness of the new results, and therefore of software/firmware which is producing the updated algorithm. Such checking could prevent catastrophic failure of the new system due to programming bugs; once the system has been satisfactorily tested in the real environment, the updated system could become the old, or master, system and a new update begun. By this process, similar to that used in bringing up a new version of an operating system, a graceful transition to the improved capability is possible.

Parallelism Applied to Dissimilar Tasks -- When parallelism is applied to dissimilar tasks, much faster response is often possible (if overhead could be neglected, the completion rate for the set of tasks would be limited only by the slowest component of the set). Depending upon the organization of how those tasks are to be applied to the data stream, two architectures are suggested for implementation of the parallelism. If all data passes through the tasks in a fixed order, a pipeline architecture is appropriate, with each processor working on a different data element in turn, with all processors working simultaneously once the pipe is full. This architecture has a very simple control structure and, once full, achieves an output rate limited by the slowest component task. Conversely, if the tasks do not have such an organization with respect to the data, they can be divided into independent (or, more practically, quasi-independent) processes which are then executed simultaneously on different parallel processors. The usefulness of this approach is limited both by the control overhead necessary to coordinate the processes and by the difficulty of partitioning the tasks into independent

processes, but can sometimes achieve completion rates which approach that of the slowest task.

Alternatively, and of more interest in DDP design, is that more capability can be built into the system while still maintaining the same real-time response. That is, the time remaining in the real-time cycle, which was made available through parallel task execution, can be made available for other system functions. This implies that the designer can insert capability for testing and control, while still meeting the response requirements, which can impact favorably on V&V, reliability, graceful degradation, etc.

Such partitioning of the set of tasks into concurrent processes also makes feasible distributing the data-processing capability and allows the exploitation of smaller hardware components. This design approach can improve survivability by allowing geographical distribution of the system components. Availability can also often be improved since if the same hardware component is replicated in a number of different (parallel) portions of the system, it may be cost effective to provide a spare. This spare part approach is rarely cost effective when large computers are the element for which a spare would have to be provided.

5.3.3 Control and Data Base Formalism

The formalism proposed for describing control and data base distribution has a number of design advantages which should make easier the satisfaction of the design requirements vectors. First, the formalism clearly delineates the connectivity of system functions by showing explicitly both the required data transfers and the necessary control interactions. An examination of the required data transfers and control interactions will suggest which system functions should be grouped together and which can easily be separated. For example, the radar scheduling function has such high interaction with the

radar that they are natural candidates for grouping. Thus the explicit description provided by the formalism can assist the designer in partitioning tasks into processes in such a way that interprocess communication is minimal.

Moreover, since the interactions are all shown in the abstract representation, the decision of how to implement a particular data transfer (e.g., shared [common] memory versus message handler) or control function (e.g., formal monitor versus semaphore) is made explicitly, not implicitly as is so often true today. Thus the designer will often be faced with a clear choice between program control and efficiency. Since program control impacts strongly on V&V, survivability, growth potential, etc., it is extremely important that the designer face these decisions as early in the design process as possible. It should be mentioned that the formalism does not address the problem of how to implement monitors in a distributed system, but only indicates the required control.

Actual implementation of monitors or other formal control devices (semaphores, message handlers, overseers[18]) implies that interprocess communication is well structured. This in turn implies good program testing and debugging capability as well as potentially enhanced fault tolerance through the addition of assertion testing, e.g., range checking of parameters, legality checking of calls, etc. Data base control can also be improved by exploiting the facility of monitors for controlling data validity, concurrency, security and privacy. Implementation of such control can greatly assist V&V and graceful degradation, in particular.

Finally, the formal structures at both the design and implementation levels make DDP much easier to effect. Since all interfaces are explicitly stated, the design decision of how to partition the system is easier to make. Moreover, if the decision is made to implement control in monitors or other thin-wire compatible form, some distribution decisions can be deferred until

quite late in the design process. The possibility also exists for dynamic redistribution of functions to achieve reliability or graceful degradation. Such options are often foreclosed in current design practice by a choice of the most efficient control structure over more general and flexible implementation. While that may well be the correct design decision, it should be made with full awareness of what the consequences are.

5.3.4 Partitioning and Distribution of Capability

Properly used, partitioning is a powerful tool to yield better system designs, and can be applied at all levels of the system. The goal is to achieve modules which are as nearly independent as possible with respect to specific requirements. The selection of module partitions is a non-trivial task, and requires maximizing the intramodule dependencies (also called module strength[19] while simultaneously minimizing the intermodule dependencies (module coupling). A module should be the smallest grouping, appropriate to the current level of abstraction which satisfies this criterion. A system whose module structure satisfies the above criterion will be much easier to maintain and change/adapt since a change in one module will have minimum effect on the others. System reliability is also improved since errors tend to be detected closer to the point of origin and to propagate more slowly. Since intermodule dependencies have been minimized, communications between modules will be more tractable, which means it will be possible to distribute capability to different physical locations. There is strong interaction between modularity and distributed capability on the one hand, and parallelism and control formalism on the other. The proper choice of modules makes parallel execution possible and facilitates the data sharing and program control between modules. The control formalism should make proper module partitioning easier by exposing all the data and control coupling. Monitor-like implementation of control then makes the physical distribution of function capability easier since much of the necessary protocol for intermodule

communication is already present. When physical distribution is provided, adequate facilities can be provided to execute the modules virtually in isolation from the rest. This design/implementation strategy can yield considerably improved survivability, reconfigurability, testability, and graceful degradation.

In designing modules and deciding how to distribute the system capability, it is important to consider all of the design requirements vectors as well as the required functional performance. It is clear that changes in survivability requirements may impact on the way capability is distributed in the system; the impact of reliability requirements is equally clear. However, all the design requirements can impact upon distribution to one degree or another. For example, one might consider a standard "production module" (designed according to the above principles to perform a particular system function), to which has been grafted a parallel test module (implementing assertion testing, etc.) and a SETS module (providing a realistic set of simulated inputs and the desired outputs). Thus the basic unit of capability that has been distributed is a complete package - the required functional capability, testing capability, and a simulated driving program. Parts of this sytem may be candidates for special-purpose, firmware-controlled processors. In particular, it is easy to visualize a microprocessor-based testing system which would run fully in parallel with the production module and would provide assertion testing at all times. In this case, a considerable increase in V&V capability and in reliability would result, with little or no performance degradation at not too severe a cost penalty. Given today's microprocessor cost and performance trends, this may well become a highly cost-effective solution to satisfy the design requirements for program testability and system V&V.

SECTION 6
ISSUES AND REQUIREMENTS

6.1 RESEARCH REQUIREMENTS

During FY 77 research in the design technology area, a "baseline" DDP design theory was developed to identify critical issues and needs for further research. The baseline theory, although incomplete as a design tool, provides a starting point for future design technology development and experimentation. In overview, the issues noted from the baseline theory are tied to determination of the following aspects:

- How to accommodate each requirement in the design
- "Best" sequence of requirements accommodation (defines major levels of design)
- How to know when design at each level has been completed and/or when iteration is required
- How to evaluate design successfulness (performance)
- Details of what to do in each activity at each level (defines levels of abstraction at each major level of design)
- "Best" sequence of activities at each level
- Details of what are the inputs/outputs of each activity
- Graphical/language (representation) forms
- Verification, validation, testing, traceability requirements
- Simulations (of design products and SETS) construction/usage

- How and when to communicate design results to the customer and system development management (documentation hierarchy).

More concisely, the really basic issues are:

1) How to determine and unambiguously represent structured* requirements (especially dealing with control and data base distribution) without falsely biasing lower-level design decisions

2) How to accommodate unstructured requirements, especially in terms of decomposition/partitioning rules, and in what sequence

3) How to define, perform, and meaningfully evaluate design technology experiments to enable demonstration and generalization of research results

4) Clear definition of design interfaces between requirements engineering, data processing engineering (DPE), and data processing architectural design (DPAD); see Figure 1

Analytic and/or experimental research is required with respect to these issues as detailed below. Since the FY 77 DDP technology research was strongly analytic, experimentally-based research and research to support future experiments is given higher priority where appropriate.

---

* The structured requirements identify functionality, action relationships, and necessary performance (port-to-port timing and accuracy). The unstructured requirements include most of the "payoffs" of DDP: survivability, growth, reliability/availability/fault tolerance, deployability, throughput, cost, etc. For further discussion, see DDP Design Technology Research Requirements Definition.

6.1.1 Structured Requirements Research

Research is required to support identification, representation, and translation of structured requirements through levels of abstraction. The near-term (FY 78) research should concentrate on the primitive entities and operations necessary and useful for describing requirements (especially distributed control and data base requirements) at each level of abstraction. Primitives must be identified to support both graphical and textual requirements representations. Based on these primitives, longer-term research (beyond FY 78) should develop associated descriptive language(s) which can support automated design processing and testing.

Issues to be resolved in the near-term research are: can the system requirements and the design decisions be described in terms of the specified primitives and in the format prescribed to delineate their interrelationships? Specifically, is the set of primitives and the format --

1) Complete. Can all the necessary system design requirements, parameters and decisions be expressed in these terms, both at the input to a design step and at the output?

2) Feasible. Is the codification of design in these terms possible with the knowledge/information available to the designers?

3) Convenient. Does the expression of the design in terms of these descriptors aid or hinder the speed or ease of the design process? To what degree?

4) Useful. Does the use of these descriptors yield a "better" design? In particular,

    a) Is there more awareness of the tradeoffs between the various design requirements vectors?

b) Does the imposed structure help to specify how to achieve the "ilities"? Can the designer see more easily how to enhance the system capability in these areas?

c) Does the imposed structure enhance the visibility of the design features? Are the factors which affect the "ilities" more visible? Can the designer see more easily the degree to which the requirements are not satisfied?

d) Does the imposed structure help to enforce consistency of design? Does it reduce design variability?

6.1.2 Unstructured Requirements Research

Research is required to support conversion of (perhaps) broad, unstructured requirements into specific "requirements vectors," which can be transformed into structured requirements. The conversion is considered to have two steps: (1) analysis/decomposition/partitioning of the requirement itself, to provide specific, unambiguous requirement components (or subrequirements); and (2) construction of a vector of elemental design responses for each independent requirement component. (A requirement vector is now conceived to be a linear array of elemental design responses; possibly, higher dimensioned arrays may be found necessary.) The issues and research requirements are:

1) Selection of the most efficient analysis/decomposition structures -- the tree structure is perhaps the simplest. Many other techniques exist for consolidating descriptive data, however. In particular, data base management studies should be applicable, especially concerning "query" languages being developed.

Priorities in developing the technology for accommodating requirements should be as follows: survivability, reliability/availability/fault tolerance, growth (change) of various system applications requirements and data processing hardware technology capabilities, throughput (performance), deployability, and cost. In the near term, each requirement is to be treated individually in detail, and collectively in a more qualitative manner. Longer-term research is to treat groups of requirement vectors.

REFERENCES

1. Hayden, D. F. and Sunshine, L., "Feasibility Study - Impact of Partitioning on Large-Scale Systems Development," (U) Bell Laboratories Memorandum, 1 April 1977, SECRET.

2. Balkovich, E., Palmer, D., and Wood, R., The Role of the Monitor as Abstraction in Designing the Control for Real-Time Data Processing Systems, IM 2124, July 1977.

3. Jones, J. C., Design Methods, Wiley-Interscience, 1970.

4. Spillers, W. R. (Ed.), Basic Questions of Design Theory, North Holland, 1974.

5. Wymore, A. W., Systems Engineering Methodology for Interdisciplinary Teams, Wiley, 1976.

6. Sage, A. P., "A Case for a Standard for Systems Engineering Methodology," IEEE Trans. on Systems, Man, and Cybernetics, Vol. SMC-7, No. 7, July 1977, pp. 499-504.

7. Bauer, F. L. (Ed.), Software Engineering -- An Advanced Course, Springer-Verlag, 1975.

8. Freeman, P. and Wasserman, A. I. (Ed.), Tutorial on Software Design Techniques, IEEE publication 76CH1145-2C, IEEE Service Center, Piscataway, N. J., 1976.

9. Spillers, W. R., "Design Theory," IEEE Trans. on Systems, Man, and Cybernetics, March 1977, pp. 201-204.

10. Peters, L. J., and Tripp, L. L., "Is Software Design 'Wicked'?", Datamation, May 1976, pp. 127-136.

11. Anderson, G. A., and Jensen, E. D., "Computer Interconnection Structures: Taxonomy, Characteristics, and Examples," Computing Surveys, Vol. 7, No. 4, December 1975.

12. Bell, T. E., et al., "An Extendable Approach to Computer-Aided Software Requirements Engineering," IEEE Trans. on Software Engineering, Vol. SE-3, No. 1, January 1977, pp. 49-60.

13. Gaulding, S. N., and Lawson, J. D., "Process Design Engineering: A Methodology for Real Time Software Development," Proc. 2nd International Conf. on Software Engineering, IEEE, Oct. 76, pp. 80-85.

14. Fitzwater, D. R., The Formal Design and Analysis of Distributed Data Processing Systems, U. of Wisc. Report CSTR279, October 1976.

15. Tsichritzis, D., et al. (Ed.), Tutorial on Data Base Languages and Systems, IEEE Catalog 76CH1144-5-C, October 12, 1976.

16. Georgiou, V., A Parallel Pipeline Multicomputer Architecture, Ph. D. Dissertation, University of California, Santa Barbara (in preparation).

17. Fishman, C., Concepts and Methods in Discrete Event Digital Simulation, John Wiley & Sons, 1973.

18. Pickens, J. R., "The OVERSEER - A Powerful Communications Attribute for Monitoring of Thin-Wire Control Structures," Proceedings of ICC, Toronto, Canada, August 1976.

19. Stevens, W. P., Myers, G. J., and Constantine, L. L., "Structured Design," IBM Systems Journal, Vol. 13, No. 2, 1974, pp. 115-139.

BIBLIOGRAPHY

1. Myers, G. J., Reliable Software through Composite Design, Petrocelli/Charter, New York, 1975.

2. Pickens, J. R., "DeBugging and Monitoring of Distributed Control Structures," Ph. D. Dissertation, Department of Electrical Engineering and Computer Science, University of California, Santa Barbara, California, May 1976.

IM-2124

APPENDIX A

THE ROLE OF THE MONITOR AS AN ABSTRACTION IN DESIGNING THE CONTROL FOR REAL-TIME DATA PROCESSING SYSTEMS

by

E. E. Balkovich
D. F. Palmer
with
R. C. Wood
University of California at Santa Barbara

July 1977

GENERAL RESEARCH CORPORATION
P.O. BOX 3587, SANTA BARBARA, CALIFORNIA 93105

Submitted to the

Texas Conference on Computing Systems

This work was supported by the Distributed Data Processing study of the Ballistic Missile Defense Advanced Technology Center as a subcontractor of the Honeywell Systems and Research Center, subcontract number P.O. 416575-FA.

ABSTRACT

One of the most critical issues in real-time system design is the specification of control. The investigation reported on here examined when, during the design process, requirements for control are first identified and how they should be specified. This examination of requirements for control is part of a larger study of the overall design process for distributed data processing systems for real-time applications. The study has primarily focused on the design outputs, recognizing that the process of design is a creative, iterative activity.

The results of the study show that the abstract concept of a monitor can be used to structure and organize the specifications for control. The result is a hierarchical, decentralized specification of control requirements amenable to analyses that can provide early verification of the design. An example is used to illustrate that such specifications of control can be implemented in a variety of ways using hardware, firmware, and software technologies.

CONTENTS

SECTION		PAGE
	ABSTRACT	1
1	INTRODUCTION	1
	1.1 Background	1
	1.2 The Problem	3
2	THE DESIGN PROCESS	5
	2.1 Control Design	6
	2.2 The Specification of Control	10
3	AN ILLUSTRATION	14
	3.1 Random Updates	16
	3.2 Ordered Updates	18
4	SUMMARY	20
5	ACKNOWLEDGEMENTS	22
	REFERENCES	23

PRECEDING PAGE BLANK-NOT FILMED

1 INTRODUCTION

1.1 BACKGROUND

The U.S. Army Ballistic Missile Defense Advanced Technology Center (BMDATC) is initiating the development of a comprehensive technology for designing real-time distributed data processing (DDP) systems [1]. The first phase of research, just completed, defined critical issues in the design of data processing systems for BMD and established a research plan for resolving the issues. This paper describes the initial research on a specific critical issue: the identification and specification of control requirements at high levels of design, prior to defining the data processing hardware.

The scope of BMDATC's research program on DDP is indicated with respect to a BMD system's life cycle in the "waterfall chart," Fig. 1.1. The DDP research program covers design engineering and implementation specification engineering, steps 2 and 3. The control-design issue addressed in this paper is part of a DDP design theory necessary to support the design engineering step. Notice that the design engineering step is preceded by a requirements definition step* and followed by an implementation specification step. (Some overlap of activities typically is necessary.)

BMD systems usually do not have the simple life cycle shown in Fig. 1.1. A BMD system's life cycle more typically consists of an iterative, evolutionary development, in which partial developments occur in the form of "paper studies," test-range versions, prototypes, etc., in advance of the primary life cycle depicted. The design theory consequently is required to be robust enough to handle more complex life cycles.

---

*BMDATC is beginning research on the requirements definition step. The program is termed Axiomatic Requirements Engineering Research.

PRECEDING PAGE BLANK-NOT FILMED

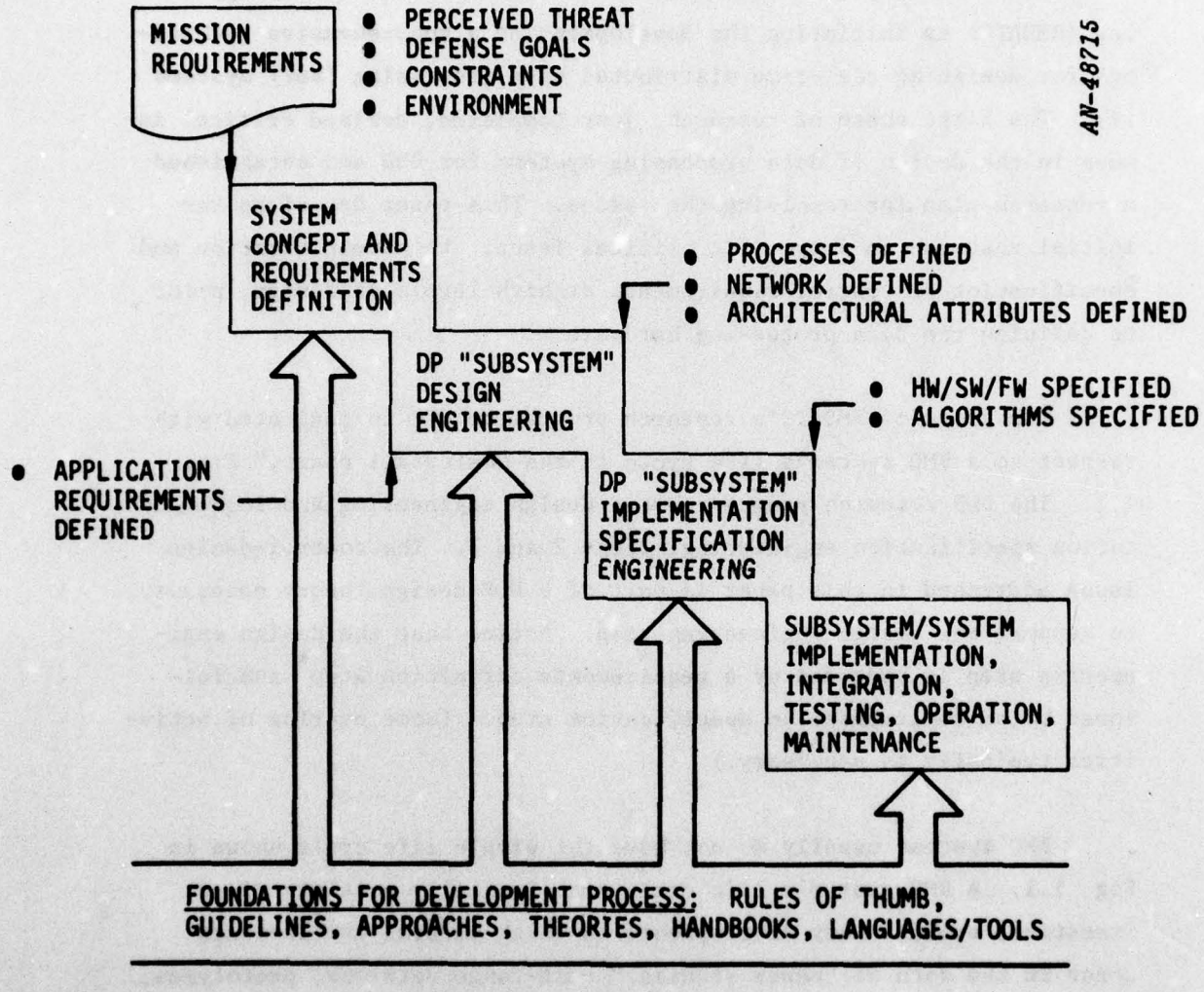


Figure 1.1. Model of Stages in BMD System Development and Use (Life Cycle)

There are many requirements which must be addressed in the design theory. These requirements relate to four aspects of design: partitioning of problem requirements, network interconnection, data base distribution, and performance determination at the various system levels. Specific topics for consideration include vulnerability, reliability, response, throughput, communications protocols, data integrity, testability, and maintainability. Control design decisions can be expected to interact with all of the aspects of design.

1.2 THE PROBLEM

As was already noted, one very important aspect of the design theory is the early identification and specification of control for the data processing system. The approach taken was to identify the design context in which control of the data processing was considered and the concepts that might be used to specify control at a very high level. By control, we mean the management, at run time, of the data processing resources: hardware, firmware, and software.

This work was influenced by earlier efforts at understanding the data processing systems analysis activities that initiate the design cycle [2] and by recent results related to the design and description of control. Perhaps the most influential concept was that of a monitor [3] and its subsequent incorporation into design and programming languages [4,5]. Our effort views this concept on a more abstract level, and uses it as the basis for specification rather than implementation.

The results of our investigation led to the specification of control in the framework of a data processing architecture expressed in terms of abstract processes and monitors. This architectural specification was derived from inputs provided during systems analysis that describe the functions that the data processing system must perform, as well as the data abstractions used in performing those functions.

Some limited experiments with this approach led us to believe that the form for specifying control requirements has considerable

utility and flexibility. The use of the monitor as an abstraction, rather than an implementation device, allows the details of the application to be suppressed and the specifications for control to be emphasized. The monitor and process concepts appear to play a powerful role in organizing, structuring, visualizing, and verifying the control features of a real-time system. We believe that early identification and specification of control will provide for independent design and development of control and application, with clearer interfaces between the two.

In the paragraphs that follow, we will discuss our use of the concepts of process and monitor in our approach to design. This discussion will address the design context in which the architecture of the data processing system can first be identified, and comment on the use of the monitor concept in the control-design specification. These general comments will then be illustrated with a simple example to show some of the interactions of control specifications and possible implementations.

2 THE DESIGN PROCESS

Our overall goal is the development of a theory (or methodology) for designing large, complex, real-time data processing systems having numerous and diverse requirements. The development of such a design theory has received much attention in recent years for general system contexts [6,7,8,9] and for general software and data processing systems [10,11]; many other references exist. The manifest result of these activities is the finding that such design problems cannot be solved in an absolute sense and, therefore, that associated design theories cannot achieve completeness nor be significantly automatable. We are dealing with a so-called "wicked" problem [7,12,13], which is partially characterized by (1) inability to define the problem completely; (2) inability to state the problem without at least partly stating (biasing) the solution; (3) no stopping rule; (4) no absolute measures of correctness or falseness; and (5) no absolute list of admissible procedures and operations. In effect, existing results indicate that design must remain largely a creative activity, having a high degree of iteration and without a simply connected, uniformly evolving set of products.

The impossibility of a fully defined and automatable design theory does not reduce the need for a design theory. It does mean that for any useful progress, early research in design theory must focus on important but tractable elements of the problem. The focus of our current research and of this paper is the determination and specification of processing control requirements at relatively high abstract levels of design, before the design and specification of hardware.

The definition of "creative steps" is a major problem in defining a design theory. To avoid the problems of stating how to be creative, our research approach directs attention to the "products" of design:

1. The Input Product (from preceding design phase)
   The form and semantics of requirements input to the control-design phase are postulated. These requirements are the main source of high-level decisions about control design.

The inputs are appropriate products of a preceding "requirements definition" phase of design, in which functions and data relationships representing the application requirements are identified.

2. The Output Product (of the control-design phase)
The form and semantics of requirements output from the control-design phase are postulated. The outputs are deemed appropriate for a following "architecture definition" phase of design, in which the hardware/software/firmware architecture is developed for implementation.

3. Internal Products (generated during control design)
Internal products and their uses to support the transformation of postulated input products to output products are derived. This structure of internal products and associated design activities serves to define a procedure for designing processing control requirements at high levels of design.

This heuristically derived design procedure is being refined through iteration of case studies and refinement of product definitions and sequences.

2.1 CONTROL DESIGN

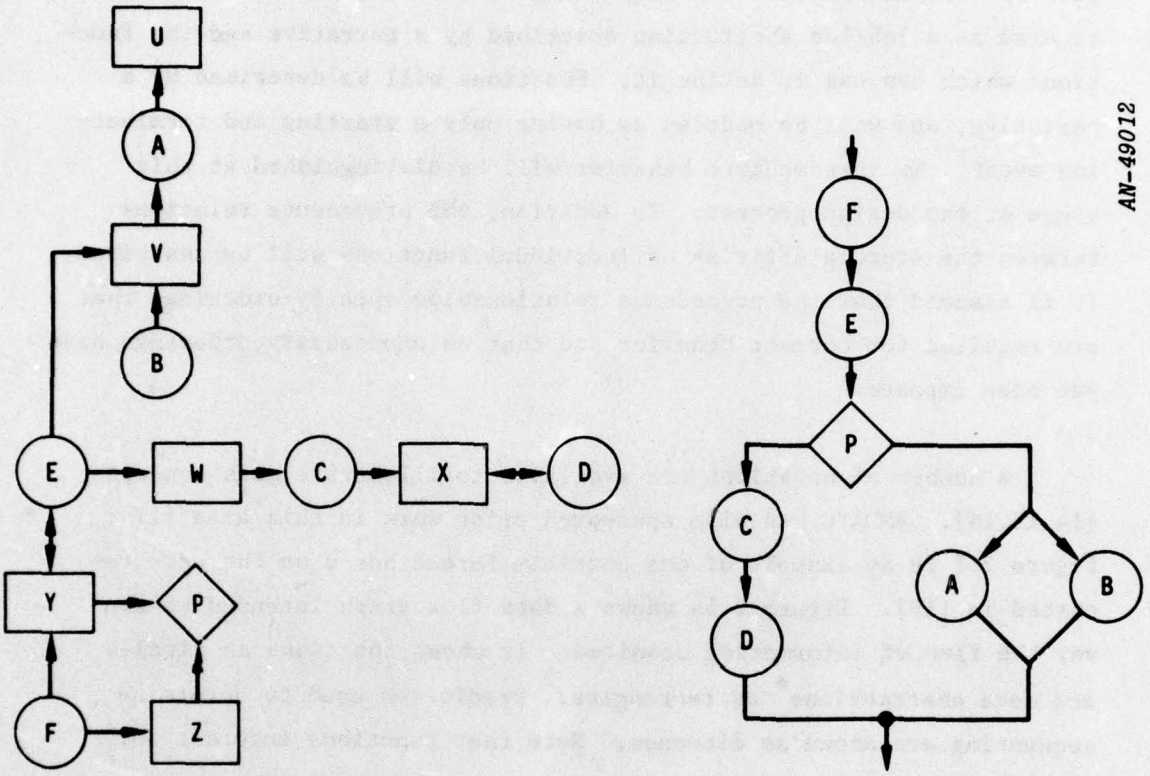
We assume that the data processing requirements, from which the control system is to be designed, are initially described in terms of functions and data abstractions. In general, we will choose to model the required computations with functions that do not retain information on any previous computations. In that case, the behavior of a function depends only on clearly delineated inputs (which may, however, have been defined by a previous use of the function). In particular, we will enforce the convention that functions which interact with one another do so only through shared data abstractions. We recognize that for reasons of efficiency or style, the implementation may differ somewhat from the functional abstraction as modeled. Our purpose is to keep control as explicit as possible throughout the design process.

At least two aspects of behavior must initially be described: data or information flow and sequencing of functions. Data will be treated as a labeled abstraction described by a narrative and the functions which can use or define it. Functions will be described by a narrative, and will be modeled as having only a starting and terminating event. No intermediate behavior will be distinguished at this stage of the design process. In addition, the precedence relations between the start and finish of individual functions will be described. It is assumed that the precedence relationships specify orderings that are required for correct behavior and that no unnecessary orderings have yet been imposed.

A number of notations are available to illustrate such concepts [14,15,16]. BMDATC has also sponsored prior work in this area [17]. Figure 2.1 is an example of one possible format based on the work reported in [16]. Figure 2.1a shows a data flow graph intended to convey the flow of information required. It shows functions as circles and data abstractions* as rectangles. Predicates used to determine sequencing are shown as diamonds. Note that functions interact only through data abstractions and that predicates are derived only from the data. Such diagrams say nothing about the order of execution of the functions. Figure 2.1b shows a corresponding precedence diagram, which describes the required order for executing functions.

Complex ordering relationships can be defined between the start and termination events of functions. We shall designate the start and termination of a function G by $\overline{G}$ and $\underline{G}$, respectively, and indicate precedence by the symbol >. For example, $\overline{G} > \underline{G}$, i.e., G must start before it can terminate. Figure 2.2 shows the possible precedence relationships between two functions.

*It should be emphasized that the data abstractions shared between functions do not imply any particular implementations such as shared data bases. They may simply represent parameters passed between procedures. The data abstractions serve to explicitly identify the coupling between functions and lead to improved designs [18].



a) Data Flow Graph

b) Precedence Diagram

A, B, C, D, E, F: FUNCTIONS

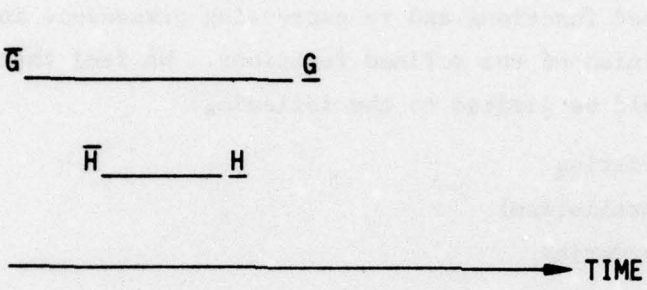
P: PREDICATE

V, W, X, Y, Z: DATA ABSTRACTIONS

Figure 2.1. Representations of Functions and Data Abstractions

A) NESTED:

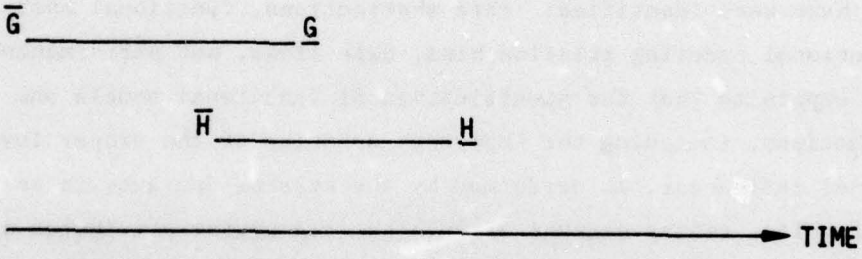
$\overline{G} \cdot\!\!> \overline{H} \cdot\!\!> \underline{H} \cdot\!\!> \underline{G}$



AN-49010

B) OVERLAPPING:

$\overline{G} \cdot\!\!> \overline{H} \cdot\!\!> \underline{G} \cdot\!\!> \underline{H}$



C) SEQUENTIAL:

$\underline{G} \cdot\!\!> \overline{H}$

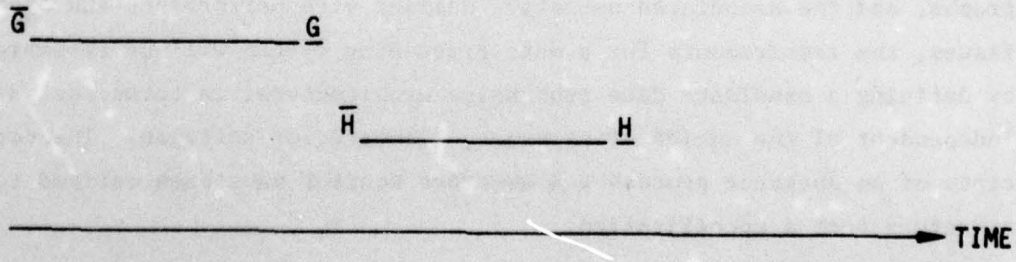


Figure 2.2. Precedence Relationships Between Start and Termination of Two Functions

AD-A047 476 HONEYWELL INC MINNEAPOLIS MINN SYSTEMS AND RESEARCH --ETC F/G 9/2
DISTRIBUTED DATA PROCESSING TECHNOLOGY. VOLUME V. APPLICATION O--ETC(U)
SEP 77 D PALMER, E BALKOVICH, R WOOD DASG60-76-C-0087
UNCLASSIFIED 77SRC69 NL

2 OF 2
AD
A047476

END
DATE
FILMED
1 -78
DDC

For a multiplicity of functions, the precedence interrelationships can become quite complex. We feel that such complex relationships can eventually be reduced to a canonical form by subdividing functions into more refined functions and re-expressing precedence in terms of the start and finish of the refined functions. We feel that such canonical forms should be limited to the following:

1. Sequential ordering
2. Fork/join (parallelism)
3. Conditional ordering

It should be noted that these forms are identical to those used in R-nets [14].

The list of concepts used in describing requirements is not complete at this point in our work. However, some of the more important primitives have been identified: data abstractions, functional abstractions, functional ordering relationships, data flows, and performance. We wish to emphasize that the specification of functional models and data abstractions, including the important question of the proper level of functional refinement, is performed by the systems designer in an iterative fashion, taking account of such desired system attributes as response time, reliability, survivability, security and integrity, testability and maintainability.

2.2 THE SPECIFICATION OF CONTROL

Given a candidate set of precedence relationships, data flow graphs, and the associated narrative dealing with performance and other issues, the requirements for a data processing system will be re-expressed by defining a candidate data processing architecture, in terms that are independent of the choice of hardware, firmware, or software. The concepts of an abstract process and abstract monitor have been defined to structure such a specification.

Performance is a critical issue for real-time systems. Generally, performance must be sacrificed to achieve other goals such as reliability

or survivability. The objective of this phase of design is to identify and describe parallelism at a high level. The implementation of parallelism requires control, and it is this observation that suggests that the requirements for control should also be identified and described at this level of design.

The notion of an abstract process serves as the basic component of the data processing architecture that is modeled (at this level of specification) as operating in parallel with other system components. A process is viewed as a model of potential parallelism, not an implementation, since no assumptions have been made about the underlying hardware, firmware, or software. Functions are assigned to processes. One or more functions can be controlled by a process.* A process is limited in that it can only cause functions to be executed sequentially. Thus control, from the point of view of a process, can be expressed simply as a flow chart. This is one aspect of the requirements for control (i.e., the ordering of functions by a process).

At this level of detail, all parallelism is between processes. However, it is quite probable that as design proceeds and hardware implementations are considered, further potential parallelism will be identified. In that case, the processes and their functions will be redefined as appropriate.

Functions will be assigned to processes by considering tradeoffs between the benefits of parallelism and the "implementation overhead" of shared data abstractions and synchronization. The examples which follow will show that this choice of assignments will depend strongly not only on the function precedence graph and the data flow graph, but also on the interface rules for the shared data abstractions as set forth in the narrative. Data abstractions will evolve from the input requirements and will also be created to coordinate processes. These latter abstractions (induced by the need for coordination) look like

*Recall that functions can be refined to a level of detail appropriate for making such assignments.

simple synchronization devices (e.g., semaphores or fork/join).

Clearly, parallel components of the system interact. It is these interactions that give rise to other aspects of control. Such interactions will be expressed and represented as abstract monitors. The monitor will be used to represent an architectural component shared between processes. This type of system component is defined by one or more mutually exclusive operations on a shared data abstraction.* The execution of each operation by a process using the monitor may cause the rescheduling of any of the processes that use the monitor.

An abstract monitor will be the basic structure for defining operations that are used to cause processes to interact. Such operations are assumed to be available to any function running under the control of a process which has access rights to the monitor which embodies those operations. The correct usage of the operations might also be specified.** In addition, the monitor will specify the rules for scheduling the execution of processes which interact through it. Thus, the monitor represents a data abstraction (which may be trivial) and control.

The task of the control designer is thus to examine the precedence graphs for sets of independent (equal-precedence) functions, which are potential candidates for parallel implementation. The identification of possible parallel processes implies the introduction of one or more induced abstract monitors to control the synchronization of these processes. Given such a possible set of processes, with their contained functions,

---

*Other control activities like synchronization can be modeled as the sending and receiving of zero-length messages through a shared buffer (a data abstraction). Thus, the single notion of a monitor will be used to identify all requirements for control. It is important to emphasize that the abstract monitor serves only in the identification and specification of control requirements and is not necessarily the implementation of choice for control. More will be said about implementation in Sec. 3.

**The use of path expressions is one possible means of describing these orderings [19,20,21].

the designer must examine the data flow graphs, in conjunction with the precedence graphs, to determine the need for abstract monitors to coordinate the use of the shared data abstractions.

Experiments using monitors [22] indicate that the abstract notions of monitors and processes, when related through an access rights graph [4], offer an excellent pictorial representation of the architecture of a data processing system. This representation has the further advantage that it is amenable to analysis. It has been used in identifying and preventing certain forms of deadlock. We therefore believe that it has considerable potential in verifying the correctness of a design at a very high level.

By designing the requirements for control in terms of these concepts, several objectives are achieved. An initial specification of requirements for the data processing system, expressed as temporal relationships, has been restructured into a hierarchical organization expressed in terms of a few simple types of system components. These simple types suppress many of the details of the application and emphasize the requirements for control. Furthermore, the requirements have been stated in a decentralized form. The requirements for control have been distributed among individual monitors that may ultimately be implemented in a distributed fashion. These requirements are related to specific data abstractions that are shared among the processes, interacting through the monitor. This has the further advantage that the specifications of control have been clearly delineated and separated from the specifications of the application. The latter are viewed as individual sequential functions. This can lead to the separate development of application and control as in SOLO [22] and possibly reduce the development time required for complex real-time systems.

3 AN ILLUSTRATION

A simple example can be used to illustrate how abstract monitors serve in the specification of control. Such an example can also be used to comment on the relationship between implementations and the specifications. The following illustration is based upon a specific interpretation of portions of Fig. 2.1 (which would be supplied as a narrative description with the figure). Suppose that the data abstraction V is a data base consisting of a number of records; function B updates records of the data base and function A uses records of the data base. Assume that A and B are the only functions that access V after initialization by the function E . In addition, suppose that the consistency of the records must be maintained, so that simultaneous updating and reading of records is not allowed. This particular interpretation is a specific case of the more general Reader/Writer problem [23] and is representative of such real-world entities as a track file in a radar application.

The specification of A and B shown in Fig. 2.1 suggests the data processing architecture shown in Fig. 3.1.* In this design, the processes P_1 and P_2 may execute simultaneously. They interact when accessing a shared data abstraction, the data base. The abstract monitor M_2 represents the control of those interactions. In addition, the abstract monitor M_1 is introduced to represent the control required to initiate and terminate the processes P_1 and P_2 . The process P_3 is shown to illustrate a process which may control activities that must proceed and succeed the execution of P_1 and P_2 . It interacts through the control prescribed by the abstract monitor M_1 .

Although this data processing architecture identifies two monitors, only one is needed. However, the two monitors are intended to clearly distinguish between control that initiates and terminates the updating and

*The notation is that of an access rights diagram suggested in [4]. These diagrams illustrate the processes allowed to share the operations provided by particular monitors. What is shown is a portion of the architecture required by Fig. 2.1. Only the functions A and B have been considered in arriving at this diagram.

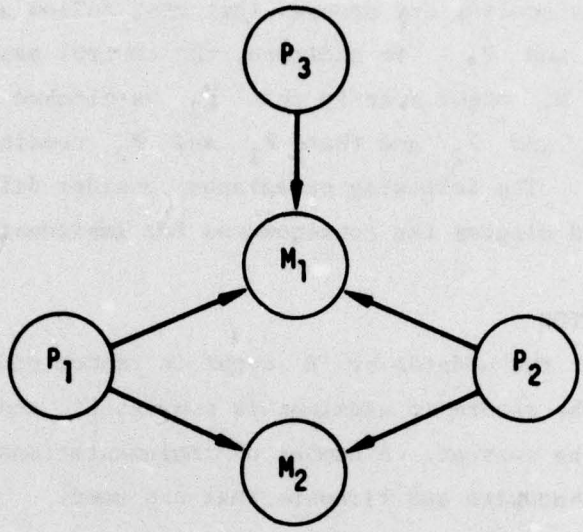


Figure 3.1. A Candidate Data Processing Architecture for the Functions A and B

use of the data base, and control that manages the use of the shared data base by the competing processes. The diagram provides an organization and a pictorial representation on which further specification of control is based. As noted in the previous section, these further specifications should define such items as the operations that the monitor makes available, the processes that have access to the monitor, scheduling of the processes, etc. The specifications require design decisions that may preclude some implementations. Of course, one should not consider any decision at this level final until its consequences have been evaluated.

For this particular example, it is the specification of control associated with access to the data base (M_2) that drives the possible solutions or implementations. The abstract monitor M_1 might specify such operations as the sending and receiving of signals that initiate the processes P_1 and P_2 and assure that both processes have been

completed before signaling any process that must follow after the completion of both P_1 and P_2. In addition, the control associated with the abstract monitor M_1 might specify that P_3 be blocked pending the completion of P_1 and P_2 and that P_1 and P_2 remain blocked until initiated by P_3. The following paragraphs consider different specifications for M_2 and discuss the consequences for implementation.

3.1 RANDOM UPDATES

Suppose that the updates by B occur in random order, and that A cares only that the record it examines is consistent, and not that the entire data base be current. A number of implementations are possible depending on the hardware and firmware that are used.

3.1.1 Control of Record Access

Suppose that M_2 specifies that the access to individual records is to be controlled. This might be accomplished by requiring an implementation of M_2 to provide operations that grant exclusive access to a requesting process on a record-by-record basis. This implies that a requesting process must request and release access rights to each record that it wishes to access. Two implementations of these specifications are possible, depending on the hardware and firmware that are ultimately selected.

If the records of V correspond to a hardware or firmware unit in memory (byte, word, or larger block for a firmware-controlled "smart" memory access system), the control represented by the abstract monitor M_2 is subsumed by the hardware. This implies that the only overhead of the shared data abstraction is that caused by memory interference.

If, on the other hand, the records of V are larger than the hardware/firmware unit, then control specified by the abstract monitor M_2 must be implemented in software. In this case, the overhead incurred in sharing the data abstraction is high if the references are frequent.

However, the system is responsive and the maximum delay time due to blocking is minimal, compared to the software approaches discussed below.

3.1.2 Control of Data Base Access

Now suppose that M_2 specifies that the access to the data base, rather than individual records, is to be controlled. This might be accomplished by defining M_2 to have operations that grant exclusive access to a requesting process that must request (release) access rights to the entire data base before (after) referencing any record. If M_2 grants access to all records of the data base and is implemented as a conventional monitor, behavior then depends on how the monitor is used.

If access to the entire data base is authorized for each use of a record, then the competing process is blocked from accessing any record, even if it is a different record from that being used. This use of the specification has delay times and overheads similar to the previous controls, but has a higher blocking probability. If the control made available by the monitor is used to grant access to a group of records rather than single records, then the control overhead is reduced from that of the previous techniques, but the potential delays are increased. Preemption is a technique that can be used to deal with unacceptable delays; however, its implementation introduces considerable overhead.

An alternative is to have the processes pass access rights to the data base back and forth in a co-routine relationship. In this case, the control both grants exclusive access to any record of the data base and blocks the process relinquishing rights to the data base. As a consequence, all parallelism between the two processes is lost. The overhead of scheduling the processes on the same processor would also probably rule out this solution. However, if these processes executed on different processors which were shared with other unrelated processes, then this co-routine solution might provide a very effective strategy for scheduling the use of the processors.

3.1.3 Sequential Execution

Yet another alternative is to specify that the functions A and B run sequentially. In this case, the data processing architecture shown in Fig. 3.1 no longer applies. The entire system can be specified in the single process, P_3, which causes A and B to execute sequentially. This eliminates the need for any control other than that provided by P_3. The monitors M_1 and M_2 vanish from the specification, as do the processes P_1 and P_2. This assumes that no other process in the architecture competes with P_3 for the use of the data base.

3.2 ORDERED UPDATES

Now suppose that the function B makes ordered updates to the data base and that the function A requires the data it reads to be current (i.e., already updated by B). This is a case of coordinating two cooperating processes with a producer-consumer relationship. The same data processing architecture as shown in Fig. 3.1 applies. However, the specification of the control embodied in the monitor abstraction M_2 is significantly different. As before, the monitor must grant exclusive access to records of the file to assure consistency. However, the scheduling of the processes for execution by the monitor must take into account the order in which the records are requested. The implementation of such a scheduling strategy strongly depends on the nature of the updates by B and their duration relative to the time required for access by A. The following two cases suggest how such an abstract monitor concept adapts itself to different hardware architectures.

3.2.1 Complete Update

If the process controlling the function B updates all records, then a possible candidate architecture is a pipeline with the control represented by the abstract monitor M_2 serving as the control of the pipe. If the times to update each record and read each record can be made identical and equal, then the control represented by M_2 can be realized by a clock, causing synchronous operation of the two processes.

Again, M_1 serves to signal the start and completion of operations in the pipe, providing for the delay of one clock time in the start of A. Suppose the execution requirements of B vary from record to record. As long as the processing time required by A is less than that required by B for all records, a similar clocked relationship, asynchronously initiated by B, can be realized.

If, on the other hand, neither assumption can be guaranteed, then the two processes must cooperate, one serving as a producer and the other serving as a consumer. The control embodied by the monitor might be realized in hardware, firmware, or software as a message buffer between the two processes, with operations advancing with the completion of both tasks.

3.2.2 Partial Update

If the function B does not update all of the records, then a pipeline architecture is not feasible. In this case, control represented by M_2 is best implemented by firmware or software and must realize a more complicated scheduling algorithm that allows the process controlling A to proceed only after assuring the record referenced by A is current. This suggests a software monitor, with the inherent overhead for control that such implementations imply.

4 SUMMARY

We have illustrated that the abstract monitor lends itself to a wide spectrum of hardware/firmware/software implementations. In fact, one question which a designer should ask is: Under what conditions (choice of algorithm, hardware, process organization, etc.) would any of the abstract monitors specified in a particular design degenerate to (or be represented by) a clock, a start-stop switch, etc? Any such case is a prime target for hardware/firmware implementation. Other situations may suggest software configurations of a simpler nature than a formal software monitor implementation. Such decisions take place at a later stage of the design process; these examples merely serve to illustrate the compatibility of this task with our form of design specification for control.

We believe that the concept of an abstract monitor is an excellent conceptual mechanism for structuring and organizing specifications for control in a data processing system. We have shown that the specifications can be implemented in a variety of ways and do not generally preclude any particular form of implementation. The resulting form of specification augments other descriptions of requirements and emphasizes the control requirements while abstracting the application. This view makes the process and functional communication explicit at a high level of design.

Although there is some rudimentary notation suitable for use in such descriptions (e.g., access rights graphs), the description language for control requirements needs considerable refinement and extension. Many of the key concepts requiring description (e.g., scheduling rules) have been identified, but the precise form of their specification within the framework of an abstract monitor has been left unspecified. At the moment, only programming languages exist. While these offer the advantage of being machine-processable, they deal with requirements at too detailed a level. One goal of continued research should be the

identification of a language that is suited for use at this higher level of abstraction. Such a language should also be machine-processable. There are elements of these concepts that already lend themselves to analysis, and we are confident that the scope of possible verification and analysis techniques can be extended. Considering the size of the systems to be described, automation of the analysis would be desirable.

5 ACKNOWLEDGEMENTS

The work described here was conducted by General Research Corporation under subcontract to Honeywell Systems and Research Center, Minneapolis, Minnesota, a BMDATC DDP prime contractor. The effort was coordinated with, and influenced by, many persons working on other tasks at Honeywell. In particular, the implementation specification engineering tasks of R. Y. Kain [24] should be acknowledged.

REFERENCES

1. C. R. Vick, J. E. Scalf and W. C. McDonald, "Distributed Data Processing for Real-Time Applications," these proceedings.

2. E. E. Balkovich and G. P. Engelberg, "Research Towards a Technology to Support the Specification of Data Processing System Performance Requirements," Proceedings of the 2nd International Conference on Software Engineering, October 13-15, 1976, San Francisco, California, pp. 110-115.

3. C. A. R. Hoare, "Monitors: An Operating System Structuring Concept," Comm. ACM, Vol. 17, No. 10, October 1974, pp. 549-557.

4. P. Brinch Hansen, "The Programming Language Concurrent Pascal," IEEE Trans. on Software Engineering, Vol. SE-1, No. 2, June 1975, pp. 119-207.

5. N. Wirth, "Modula: A Language for Modular Multiprogramming," Software - Practice and Experience, Vol. 7, No. 1, Jan-Feb 1977, pp. 3-35.

6. J. C. Jones, Design Methods, Wiley-Interscience, 1970.

7. W. R. Spillers (Ed.), Basic Questions of Design Theory, North Holland, 1974.

8. A. W. Wymore, Systems Engineering Methodology for Interdisciplinary Teams, Wiley-Interscience, 1976.

9. A. P. Sage, "A Case for a Standard for Systems Engineering Methodology," IEEE Trans. on Systems, Man, and Cybernetics, Vol. SMC-7, No. 7, July 1977, pp. 499-504.

10. F. L. Bauer (Ed.), Software Engineering--An Advanced Course, Springer-Verlag, 1975.

11. P. Freeman and A. I. Wasserman (Ed.), Tutorial on Software Design Techniques, IEEE publication 76CH1145-2C, IEEE Service Center, Piscataway, N.J., 1976.

12. W. R. Spillers, "Design Theory," IEEE Trans. on Systems, Man, and Cybernetics, March 1977, pp. 201-204.

13. L. J. Peters and L. L. Tripp, "Is Software Design 'Wicked'?", Datamation, May 1976, pp. 127-136.

14. J. B. Dennis, MIT Notes on Computation Structures, 1969.

15. E. G. Coffman (Ed.), Computer and Job Shop Scheduling Theory, John Wiley & Sons, New York, New York, 1976.

16. J. R. White and T. L. Booth, "Towards an Engineering Approach to Software Design," Proceedings of the 2nd International Conference on Software Engineering, October 13-15, 1976, pp. 214-222.

17. C. G. Davis and C. R. Vick, "The Software Development System," IEEE Trans. on Software Engineering, Vol. SE-3, No. 1, January 1977, pp. 69-85.

18. W. P. Stevens, G. J. Myers and L. L. Constantine, "Structured Design," IBM Systems Journal, Vol. 13, No. 2, 1974, pp. 115-139.

19. L. Flon and A. N. Habermann, "Toward the Construction of Verifiable Software Systems," Proceedings of Conference on Data: Abstractions, Definition, and Structure, March 22-24, 1976, Salt Lake City, Utah, SIGPLAN Notices, Vol. 8, No. 2, 1976, pp. 141-148.

20. R. H. Campbell and A. N. Habermann, "The Specification of Process Synchronization by Path Expressions," Lecture Notes in Computer Science, Vol. 16, Springer Verlag, 1974.

21. A. N. Habermann, "Path Expressions," Department of Computer Science, Carnegie Mellon University, Pittsburgh, Pennsylvania, 1975.

22. P. Brinch Hansen, "The Solo Operating System: Processes, Monitors, and Classes," Software-Practice and Experience, Vol. 6, No. 2, April-June 1976, pp. 165-200.

23. P. J. Courtois, F. Heymans and D. L. Parnas, "Concurrent Control with 'Readers' and 'Writers'," Comm. ACM, Vol. 14, No. 10, October 1971, pp. 667-668.

24. M. G. Gouda, et al., "Towards a Methodology for Distributed Computer System Design," these proceedings.