THE FORMAL DESIGN AND ANALYSIS OF DISTRIBUTED DATA-PROCESSING SYSTEMS

by

D. R. Fitzwater

Computer Sciences Technical Report #322

August 1978

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

by

D. R. Fitzwater

ABSTRACT

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

A hierarchical design schema will be developed within which formal representations and analyses can be defined and the required solutions can be found.

This report describes further work (based on CSTR 295) in DDP design, real-time systems, evolutionary processes and requirements analysis. It also presents a systematic development of the required design theory, with some examples of its use.

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

TABLE OF CONTENTS

1. Executive Overview

1.1 Payoffs..... 6

1.2 Research Approach..... 7

1.3 Specification Properties..... 8

1.4 Specification Language..... 10

1.5 Methodology..... 10

1.6 Design Principles..... 11

1.7 Future Work..... 13

2. Introduction

2.1 Technical Quotation..... 15

2.1.1 Research Area..... 15

2.1.2 Overall Approach..... 25

2.1.3 Critical Issues..... 28

2.1.4 Work Plans..... 29

2.2 Organization..... 31

3. A Theory of Design

3.1 Research Approach..... 33

3.1.1 Discrete Processes..... 34

3.1.2 System Development..... 35

3.1.3 Design Theory Development..... 38

3.1.4 Conclusions..... 53

3.2 Distributed Data-Processing Systems..... 55

3.2.1 Development Payoffs..... 55

3.2.2 DDP Characteristics..... 59
3.3 Specification Properties..... 64
3.3.1 Formality..... 64
3.3.2 Consistency..... 69
3.3.3 Effectiveness..... 70
3.3.4 Homogeneity..... 73
3.3.5 Modularity..... 76
3.3.6 Informal Extensibility..... 78
3.3.7 Distributed..... 79
3.3.8 Generality..... 83
3.3.9 Conclusions..... 83
3.4 Specifications..... 88
3.4.1 Nondistributed Systems..... 89
3.4.2 Distributed Systems..... 91
3.4.3 Specification Language..... 101
3.4.4 Computer Assisted Formal Engineering Laboratory (CAFEL)..... 133
3.4.5 Specification Analysis..... 138
3.4.6 Simulation..... 144
3.4.7 Conclusions..... 146
3.5 Methodology..... 148
3.5.1 Introduction..... 148
3.5.2 Approximation..... 149
3.5.3 Decomposition/Integration..... 152
3.5.4 Elaboration..... 156
3.5.5 Optimization..... 157
3.5.6 Evolution..... 159

3.5.7 Conclusions..... 161

3.6 Design Principles..... 163

3.6.1 Formal Methodology..... 163

3.6.2 Closure..... 165

3.6.3 Unbounded Resources..... 167

3.6.4 Optimization..... 168

3.6.5 Evolutionary Adaptation..... 171

3.6.6 Conclusions..... 172

4. Patient Monitoring Example

4.1 A Development Process..... 175

4.2 Idealized Behavior..... 179

4.3 Process Approximation..... 182

4.4 Closed Loop System..... 188

4.5 Process Sharing..... 193

4.6 Process Elaboration..... 197

4.7 A Storage Model..... 199

4.8 Process and Storage Partitioning..... 203

4.9 Conclusions..... 208

5. Functional Specification of a Microprocessor

5.1 Introduction..... 210

5.2 Notation and Organization..... 213

5.3 Basic Definitions..... 217

5.4 The Central Processor..... 220

5.4.1 Overview..... 220

5.4.2 Definitions for Interprocessor Interaction and Instruction Decoding..... 222

5.4.3 Definitions for Machine Instructions..... 229

5.5 Memory Processors..... 244

5.5.1 Overview..... 244

5.5.2 Memory Processor Definitions..... 246

5.6 The Peripheral Interface Adapter..... 247

5.6.1 Overview..... 247

5.6.2 Peripheral Interface Adapter Definitions..... 249

6. Conclusions and Future Work

6.1 Conclusions..... 252

6.2 Future Work..... 255

6.2.1 Research Area..... 255

6.2.2 Overall Approach..... 263

6.2.3 Critical Issues..... 264

6.2.4 Work Plans..... 266

Appendix A - Specification of Asynchronous Interactions Using Primitive Functions..... 269

Appendix B - Petri Net Models..... 292

Appendix C - Formal Syntax Definitions..... 310

Appendix D - An Investigation of Digital System Equivalence in the Context of a Comprehensive Design Theory..... 313

Appendix E - Index to Definitions in Section 3..... 360

THE FORMAL DESIGN AND ANALYSIS OF DISTRIBUTED DATA-PROCESSING SYSTEMS

1. Executive Ov.

We will start this report with a general structure and results of this res. research has been carried out under Contract 0080 with the Ballistic Missile Defense Systems part of a continuing program to develop the use of . data processing architectures in the solution of compl. ,e scale real time defense systems.

1.1 Payoffs

The major mission payoffs being addressed (however indirectly) are life cycle costs, mission confidence, rapid development, and evolutionary deployment. These payoffs are strongly impacted by such properties as the following:

- The predictability of development systems
- The efficiency of development systems
- The reliability of development systems
- The adaptability of change of both development and application systems
- The testability of application requirement, design, and realization behaviors
- The reliability of application requirements, designs, and realizations.

The essential attributes of distributed data processing (DDP) systems include

- Distribution - both logical and physical
- Architectural domain - relatively unconstrained

- Problem adaptability - fitting architecture to problem
- Modularity - independently designed and tested units.

We can thus augment our list of critical properties above by adding simplicity as a property. We must find a way to exploit the modularity of DDP to obtain overall simplicity. Without this overall simplicity, DDP enormously complicates our development processes. The current state of development methodology is barely adequate to deal with centralized data processing systems. We must find a way to both generalize and simplify the development problems.

1.2 Research Approach

The critical properties of simplicity and testability dictate that we provide a formal structure to the development process that will prevent large classes of errors and avoid worst case situations that are too complex to resolve. Large scale development processes require substantial automation of the verification and validation tests for any confidence to be established. Even simple changes may introduce undetected inconsistencies, unless they are carried out by tested and automated procedures.

We have developed a generally applicable research procedure that, if followed, will produce a formal development methodology. The procedure is driven by a set of formal specification properties selected to have a high impact on system payoffs. A formal specification language is then defined so that the required properties are testable. A set of analysis

and simulation tools is then developed to provide feedback to a designer. Next, we develop a set of designer selected and guided procedures for transforming specifications. By the use of these automated procedures a designer may start with a very high level (and general) specification (requirements) of a desired system and produce (as a sequence of such procedure applications) a final low level (and detailed) specification (suitable for analysis, simulation, and implementation. And finally, we produce some general design principles that may usefully guide a designer in the selection of the procedure sequences so as to carry out a more reliable and efficient application system development.

1.3 Specification Properties

An informal characterization of the formal specification properties required for structuring a DDP development are given below.

Formal: A specification is formal if it is an abstraction (i.e., a thing representing only a certain set of properties, instead of its literal self) such that its represented properties can be specified precisely. We may thus automate the analysis and transformations of specifications.

Consistent: A specification is consistent if it specifies a unique formal system that is implementable. All errors such as contradictions, omissions, or impossible constraints are automatically detectable.

Effective: A specification itself may be used to generate automatically a simulation model. Experiments using the simulation may be used to display and analyze the behavior of the specified system.

Homogeneous: Every abstraction of a system must also have a formal specification. The same specification language may be used throughout the development process.

Modular: A specification is modular if it can be partitioned into identifiable components which could be replaced by compatible components, while producing only local and predictable changes in the specified system.

Informal Extensibility: A specification may contain informal (formally uninterpreted) attributes. All relevant information, not currently included in the specification's formal properties, may be associated with specification components. Where our formal methodology does not help, it must not hinder a designer using any informal techniques.

Distributed: A specification must be able to define systems composed of asynchronously interacting subsystems. We can then design and study the properties of the subsystems in isolation, knowing that their integration will not produce new or unexpected behavior.

Generality: A specification language is general if there is a specification in the language for every distinct formal system. We may deliberately constrain generality in order to eliminate untestable systems.

This abstract set of properties for specifications are

critical for creating and comparing formal design theories that will enable the achievement of the potential DDP payoffs.

1.4 Specification Language

We have developed a formal model for asynchronously interacting processes and an interpretation of such processes as system generators. Thus we may design processes as implicit system designs. The simulation of a process will display the computations of the generated system.

The process model is functional (in the mathematical sense of the word) and algebraically specified. Thus the specification language is a subset of conventional algebraic notation with the addition of some new abstract models for asynchronous interactions called "exchange functions". The use of exchange functions allows a practical and explicit design of interacting processes (and corresponding systems).

The algebraic specification language is sufficiently constrained so that the specification properties described above are automatically testable.

The specification language was used to start the design of a prototype computer assisted formal engineering laboratory that will provide the formal methodology to a system designer.

1.5 Methodology

Our formal methodology consists of a set of procedures for generating specifications, a set of automatable transformation and analysis tools, and a set of rules for their use.

We have constrained our procedures to support a homogeneous development process, starting with very abstract specifications (requirements) and ending with very detailed specifications (hardware and software designs).

In effect, the set of methodology procedures defines a space of development processes. Each development process will consist of the application of some sequence of methodology procedures to an initial specification. The results of each procedure define the new state of the development process. We can thus formally specify and study such development processes themselves, as well as those of the systems being developed.

The methodology currently includes procedures for starting, changing, decomposing, and integrating development processes. As well as procedures for elaborating and optimizing a specification. The formal nature of the specifications and the development process provides a high degree of traceability for design decisions and control of the locality of changes.

1.6 Design Principles

We have done the least work on this area since such principles must be tested by practical experience to gain credence. Indeed, most such principles arise from design experience with a methodology, and we have little such experience as yet. We can, however, identify some general principles that can be plausibly justified a priori. They must still be tested by experience.

critical for creating and comparing formal design theories that will enable the achievement of the potential DDP payoffs.

1.4 Specification Language

We have developed a formal model for asynchronously interacting processes and an interpretation of such processes as system generators. Thus we may design processes as implicit system designs. The simulation of a process will display the computations of the generated system.

The process model is functional (in the mathematical sense of the word) and algebraically specified. Thus the specification language is a subset of conventional algebraic notation with the addition of some new abstract models for asynchronous interactions called "exchange functions". The use of exchange functions allows a practical and explicit design of interacting processes (and corresponding systems).

The algebraic specification language is sufficiently constrained so that the specification properties described above are automatically testable.

The specification language was used to start the design of a prototype computer assisted formal engineering laboratory that will provide the formal methodology to a system designer.

1.5 Methodology

Our formal methodology consists of a set of procedures for generating specifications, a set of automatable transformation and analysis tools, and a set of rules for their use.

We have constrained our procedures to support a homogeneous development process, starting with very abstract specifications (requirements) and ending with very detailed specifications (hardware and software designs).

In effect, the set of methodology procedures defines a space of development processes. Each development process will consist of the application of some sequence of methodology procedures to an initial specification. The results of each procedure define the new state of the development process. We can thus formally specify and study such development processes themselves, as well as those of the systems being developed.

The methodology currently includes procedures for starting, changing, decomposing, and integrating development processes. As well as procedures for elaborating and optimizing a specification. The formal nature of the specifications and the development process provides a high degree of traceability for design decisions and control of the locality of changes.

1.6 Design Principles

We have done the least work on this area since such principles must be tested by practical experience to gain credence. Indeed, most such principles arise from design experience with a methodology, and we have little such experience as yet. We can, however, identify some general principles that can be plausibly justified a priori. They must still be tested by experience.

Formal Methodology:

A designer should maximize the use of a formal methodology and minimize introduction of informal steps in a formal development process. This principle will produce a maximum payoff of the design methodology.

Closure:

System specifications should be a closure (i.e., defining both controlled and controlling processes) and formal decompositions should be used to factor development processes. The formal analysis and testability of systems is greatly enhanced when both the control system and the environment being controlled are formally specified.

Unbounded Resources:

A system should first be designed, without conflicts for implementing resources, to a testable level until we have a satisfactory design in all other respects. This principle greatly simplifies the initial design, and ensures that the specified system could work, even if over-designed with respect to some performances.

Optimization:

An over-designed specification may be optimized by finding an equivalent specification in which redundant resources have been eliminated by introducing resource contention for the remaining resources. Thus the performance may be degraded by such contention in just those places where overperformance has been identified.

Evolutionary Adaptation:

At each step of a development process, make only those design decisions that are required in order to delegate all of the rest of the design decisions to subsequent steps. This provides maximum freedom of (re-elaboration) changes with minimum scope of consequences. Hindsight is better than foresight. This principle minimizes need for foresight.

The area of design principles is fruitful and little explored. Our formal development methodology will make it possible to formulate such principles precisely and to validate their use. The generality of our design theory makes such research investment and experiments worthwhile.

1.7. Future Work

Given the scope of this research effort, it should not be surprising that it is only part of a continuing research project and not limited to this research contract alone.

Formal Design Theory:

We plan to extend our formal design theory and use it as a basis for comparisons and integration of other methodologies. In particular, we will explore the integration of our DDP design concepts into the BMD RSL/REVS Methodology.

We plan to extend our formal design attributes into dynamic performance analysis. This will involve creation of a basic theory of resource management.

Computer Assisted Formal Engineering Laboratory (CAFEL):

We plan to complete the design and implementation of a

Formal Methodology:

A designer should maximize the use of a formal methodology and minimize introduction of informal steps in a formal development process. This principle will produce a maximum payoff of the design methodology.

Closure:

System specifications should be a closure (i.e., defining both controlled and controlling processes) and formal decompositions should be used to factor development processes. The formal analysis and testability of systems is greatly enhanced when both the control system and the environment being controlled are formally specified.

Unbounded Resources:

A system should first be designed, without conflicts for implementing resources, to a testable level until we have a satisfactory design in all other respects. This principle greatly simplifies the initial design, and ensures that the specified system could work, even if over-designed with respect to some performances.

Optimization:

An over-designed specification may be optimized by finding an equivalent specification in which redundant resources have been eliminated by introducing resource contention for the remaining resources. Thus the performance may be degraded by such contention in just those places where overperformance has been identified.

Evolutionary Adaptation:

At each step of a development process, make only those design decisions that are required in order to delegate all of the rest of the design decisions to subsequent steps. This provides maximum freedom of (re-elaboration) changes with minimum scope of consequences. Hindsight is better than foresight. This principle minimizes need for foresight.

The area of design principles is fruitful and little explored. Our formal development methodology will make it possible to formulate such principles precisely and to validate their use. The generality of our design theory makes such research investment and experiments worthwhile.

1.7. Future Work

Given the scope of this research effort, it should not be surprising that it is only part of a continuing research project and not limited to this research contract alone.

Formal Design Theory:

We plan to extend our formal design theory and use it as a basis for comparisons and integration of other methodologies. In particular, we will explore the integration of our DDP design concepts into the BMD RSL/REVS Methodology.

We plan to extend our formal design attributes into dynamic performance analysis. This will involve creation of a basic theory of resource management.

Computer Assisted Formal Engineering Laboratory (CAFEL):

We plan to complete the design and implementation of a

prototype CAFEL that will support the application of our formal design theory to actual development process experiemnts.

DDP Development Experiments:

We plan to use our design theory to support DDP design experiments being carried out by the BMD Advanced Technology Center. Ultimately, these experiments will test the validity of our development methodology and quantitatively assess the resulting mission payoffs.

2. INTRODUCTION

The work described in this report is a continuation of the previous work on contract DASG-60-76-C-0080 under modification P0004. The following technical proposal was prepared in response to RFQ DASG60-77-Q-0077 and the resulting contract was awarded in January 1977.

2.1 Technical Quotation

2.1.1 Research Area

The system development process is currently supported by an ad hoc methodology based on informal (English text) specifications. The informal nature of the specification restricts the scope and effectiveness of potential methodology tools. Some systems do get developed in spite of difficulties, and we must accept the fact that our research in methodology should broaden the range of options in the development process rather than specify a uniquely optimal process for all systems designers and customers. This section will discuss the nature of the requirements in the SOW of the RFQ.

2.1.1.1 Background (SOW 1.0)

The RFQ is for a continuation and extension of the current work on contract DASG-60-76-C-0080.

2.1.1.1.1 BMD Systems

A good review and characterization of BMD systems has been given by Davis and Vick.[1] In particular we quote the following seven methodology requirements from that paper:

-- Data Processing Description Capability. The system must allow for inclusion of data processing limitations early in the development cycle. This must include the means for assessment of data processing induced system limitations (e.g., processing delays and inaccuracies) as well as the ability to provide accurate estimation of data processing hardware requirements, and support tradeoffs between alternative approaches.

-- Requirements Orientation. Requirements approaches must be developed which insure means for stating the required processing without the inclusion of unwarranted design detail; insure unambiguous communication of intent; provide a means to validate requirements; insure their feasibility; and be responsive to the invariable change.

-- Design. The software design process must provide a means for earlier error detection, rapid modification, and designed-in reliability. The approach must insure the production of a highly reliable modular product

[1] C. G. Davis and C. R. Vick, "The Software Development System," Proceedings of the 2nd International Conference on Software Engineering, October 1976.

which will minimize the life cycle costs.

-- Automation. The system must possess as much automation as possible in every phase of software development. The aids should be such that they provide maximum utilization of the thoroughness of the computer to eliminate many sources of human error.

-- Management. The system must consist of well-defined phases containing intermediate milestones which provide for measurements and evaluation of progress. Techniques must be devised which allow a priority costing and scheduling based upon a defined, structured approach to development.

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

-- Structured Decomposition and Development. There must be a technology which forces the problem to be stated and structured at a high level, analyzed at that level and then allows the developer to proceed with the addition of detail in an orderly, defined and measurable fashion. This must proceed from early system definition through code delivery in a traceable and flexible manner.

This technology must assure maximum designed-in reliability in the development cycle.

The intrinsic severity of typical BMD system requirements complicates the development process to a nearly unmanageable extent. This complexity poses serious problems in meeting the methodology requirements quoted above. When intrinsic gate times limit performance and force non-systematic "shoe-horning" of an application onto a centralized data processor, the ability to meet or test system requirements is seriously jeopardized.

2.1.1.1.2 Distributed Data Processing

A major potential advantage of DDP lies in the relaxation of the gate time bounds on performance via increased parallelism. If an arbitrary n-system "solution" is used, the development process may become n-factorially more complex. This extra burden would jeopardize the possibility of success in the developmental process. Instead, we can spend some of the DDP performance potential on simplicity. In the limit (probably unattainable) an n-system solution could become one-nth as complex as a one-system solution. In order to obtain this simplicity we must accept sufficient requirement and design constraints (laws) so that the development process becomes simpler and so that requirements can be (and can be shown to be) met.

2.1.1.1.3 Formal Specifications

The acceptance of design laws requires that we can test resulting specifications for their consistency with those laws. This is not possible if the specifications are informal. The addition of design laws thus requires increased formalization of the system specifications. This is not enough, however.

We cannot test arbitrary (even formal) system specifications for most properties of interest. The potential (and undetectable) worst case problems will defeat our attempts to analyze and transform specifications. Our proposed methodology must support developmental processes that generate only "well-behaved" specifications that can be tested and implemented.

2.1.1.1.4 Methodology

The enforcement of such laws in the development process for large complex systems is impractical unless such enforcement can be automated in specification language design and analysis. The avoidance of worst case system specifications will frequently require an automation of the development process steps that produce that specification. This does not imply an elimination of the designer. For example, an automated tool could generate only canonical structures that are well-behaved as a service to designers searching for a solution. The designer would guide both generation and selection, yet his choices would automatically obey the required laws.

2.1.1.1.5 Current Status

We are currently working under Army Research Contract DASG-60-76-C-0080 with the Ballistic Missile Defense Advanced Technology Center, Huntsville, Alabama. These problems have been addressed in that contract. A special report[2] on this work is available and a final report is in preparation.

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

-- Developmental process.

A discrete process model has been developed.

-- Requirements specification.

A top-down derivation of some essential requirements on requirements specification has been developed. Some typical and critical steps have been identified as candidates for methodology improvements.

-- Formal specifications.

A formal functional specification language and interpretation has been developed. This formalism allows specification of asynchronously interacting systems and processes at varying levels of abstraction and appears to be a suitable vehicle for developing our methodology.

[2] D. R. Fitzwater, "The Formal Design and Analysis of Distributed Data Processing Systems," Computer Sciences Department Technical Report CSTR279, October 1976.

-- Design process.

Some critical steps have been identified and a top level characterization of the process has been obtained. The crucial aspect is "most local" testability of design decisions.

-- Critical properties of specifications.

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

-- Top down methodology development.

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

2.1.1.2 Objectives (SOW 2.0)

The long term objectives in SOW 2.0 are reasonably clear and need little interpretation. The major problem lies in quantifying such goals and measuring progress toward them.

The short term objectives are designed to build on our previous results and extend the domain of specified system

properties, testable at the specification level, to critical problems identified by the previous work.

2.1.1.2.1 Define DDP Design Theories

The design process that accepts abstract functional specifications and produces DDP process specifications must be elaborated to provide the basis for the specification of methodology tools to support it. This study must be top-down and systematic with respect to the selected properties. This objective must be met prior to tool development and prototype demonstrations required in the long term objectives.

2.1.1.2.2.2 Define Critical Real-Time Properties

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

2.1.1.2.3 Conduct Requirements Analysis

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

2.1.1.2.4 Define Evolutionary Processes

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

2.1.1.2.5 Identify Potential BMD Payoffs

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

2.1.1.3 Research Requirements (SOW 3.0)

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

2.1.1.3.1 Distributed Data Processing Design.

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

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

2.1.1.3.2 Real-Time Systems

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

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

2.1.1.3.3 Evolutionary Processes

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

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

2.1.1.3.4 Requirements Analysis

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

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

2.1.2 Overall Approach (SOW 3.0)

The general approach described by the previous contract reports will be followed in this research work and appears to

be a satisfactory basis for this work. Because of the complex nature of this research, details of the approach must be produced, tested, and elaborated during the work. We can identify some of the required tasks as discussed below. Undoubtedly others will also be required.

2.1.2.1 Formal Specifications (SOW 3.1-3.4)

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

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

2.1.2.2 Distributed Data Processing Design (SOW 3.1)

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

2.1.2.3 Real-Time Systems (SOW 3.2)

There appear to be three aspects to this task. The first is the analysis of critical reflex paths in a specified system to demonstrate that they are bounded and provide a model of performance coupling with other paths. The second is to simplify the performance surface (e.g., to ensure that it is convex) and thus simplify real-time testing. The third is to find sufficient design laws to minimize the required real-time testing. We plan to pursue all three possibilities.

2.1.2.4 Evolutionary Processes (SOW 3.3)

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

2.1.2.5 Requirements Analysis (SOW 3.4)

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

2.1.3 Critical Issues (SOW 3.0)

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

2.1.3.1 Distributed Data Processing (SOW 3.1)

- -- Equivalence. What are equivalence classes? Which members are optimal with respect to performance requirements?
- -- Distributed data and control models. How suitable are the proposed virtual networks? What canonical structures are sufficient?
- -- Decomposition/integration models. Particularly into application and virtual operating systems such that path coupling via resource contention is modeled.
- -- Functional simulation. What techniques are applicable and how can they be exploited?

2.1.3.2 Real-Time Systems (SOW 3.2)

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

2.1.3.3 Evolutionary Processes (SOW 3.3)

- -- Definition of domain of evolutionary processes.
- -- Localization and delegation of design decisions.
- -- Specification analysis of required process invariances.
- -- What are required process invariances?

2.1.3.4 A Requirements Analysis (SOW 3.4)

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

2.1.4 Work Plans

We plan to carry out the work at the University of Wisconsin utilizing the normal facilities and equipment of the university. No subcontractors or consultants will be employed.

2.1.4.1 Computer Usage

The use of computers is anticipated in the following areas:

- -- Date base management.
- -- Document and report generation.
- -- Prototype tool development.
- -- Analysis experiments.
- -- Utilization of BMD software development systems for:
  - -- Experience
  - -- Comparision
  - -- Embedding evaluation

A portion of this work must be done at the ARC due to the availability of the BMD systems. Part of the remainder can be done either with BMD or university computers. Some data base and document generation work may be best carried out on university computers.

We plan to utilize university computers, terminals, and DAIN lines to access the BMD computers.

The majority of the computer work will be carried out next summer after the critical experiments and tools have been specified. Estimation of computer usage at this time is very difficult, but we anticipate that ten hours of CDC 7600 time at the ARC would not be unreasonable. Flexibility in this figure would be potentially helpful.

2.1.4.2 Research Effort

This technical quotation is for an increased level of support to pursue the identified issues. The staff for the spring semester will be increased but limited by the availability of qualified candidates. Maximum staffing will be reached this summer coincidental with the increased computer program developments. I plan to work on this contract on a half-time basis during the spring semester, full-time during the summer, and half-time in September for the fall semester. The majority of the staff will be highly qualified graduate students working on (or preparing for) Ph.D. theses in the

area of this contract, while finishing their studies. The quality of work will be such that the major results should (and will) be published in appropriate technical journals, and meetings. A small amount of clerical help will also be required.

2.? Organization

The major results of the research effort are presented in section 3 on "A Theory of Design." We then present a sequence of examples of formal specifications of a "patient monitoring system," and a complete formal specification of a microprocessor system. We then conclude with a summary and an outline of future work.

3. A Theory of Design

The major results of our work are described in this section with references to more detailed discussions in appendices. We develop a theory of design for distributed data processing systems suitable for complex, large scale, real-time ballistic missile defense systems.

The developed design theory is, of course, not complete. It is sufficiently developed to have a major impact on systems development and software engineering technology.

The developed design theory does provide a firm theoretical basis both for further development of a viable science of design and for further specializations for ballistic missile defense system applications.

3.1 Research Approach

The task of creating a suitable system design theory is even more complex than using it to support a system development process. We have no real hope of formalizing the creative activities involved in producing a design theory. We can, however, provide some useful factorizations of the theory developments process into simpler and solvable subproblems.

We will introduce and informally describe a useful decomposition of a research process suitable for developing design theories for many classes of application systems. We will introduce a simple model of system development processes, distinguish between system development and theory development problems, and present a procedural description of a plausible research process. The validation of this research process is of course in its application. The process described seems to have broad applicability and has been used in developing a theory of distributed data processing system design.

We will start with the postulate that our design theory will be a formal theory. That is, it will be possible to decide what the design principles are and formally analyze what they do to a design. This means that the principles will be well defined, teachable, and transferable between both designers and projects.

We will also postulate that we want a design theory suitable for large scale (more than a dozen developers)

development processes. The scale requires substantial design automation to prevent, detect, or correct designer errors. The extent to which large scale automation can be carried out is directly constrained by the formalism of the design theory. We will postulate that the specification must be modular in order to control complexity and support practical automation.

3.1.1 Discrete Processes

First we will introduce the concept of an abstract discrete process and distinguish between system development problems and theory development problems.

A discrete process is a state space and a relation between elements of that state space. An evaluation of the successor relation applied to a subset of the process state space and producing a subset of the process state space is called a process step. A discrete process may thus be interpreted as a generator of a set of computations.

A computation is a sequence of state space elements. The relation of a discrete process may be applied to the first member (an initial state) of a computation to generate a second member. The relation may also be reapplied to each new member of the computation to produce a successor member. This successor state generation is a computation step.

For example, if the state space N is the set of integers and the successor relation R is "increment by any prime number,"

the discrete process (R,N) will generate (among others) the computations 3, 6, 11, 18, 21, . . . and 3, 5, 7, 9, 11, ...

Both the state space and the successor relation may be defined in terms of primitive components whose nature is not formally specified. These primitive components need not be simple and may subsequently be elaborated into more elementary primitives. Thus we may also define abstractions of discrete processes that are also discrete processes defined in terms of more abstract primitives.

The concept of discrete processes can be extended to include asynchronously interacting processes and to apply them to both system and development processes. For this discussion we will use only the simpler discrete processes defined above.

It is important to remember that we distinguish between research, development, and system processes. A research computation corresponds to the development of a design theory. A development computation corresponds to the development of a particular system design. A system computation corresponds to the operations of an implemented system design.

3.1.2 System Development

A system development process might be quite unstructured and formalizable only in terms of the final design specifications. We could still abstract all such processes as discrete processes in a trivial fashion. For example, any development process specified as (D,S), where D is a primitive

relation and S is the null set unioned with all possible design specifications, generates only computations of the form { }, "final specification." Only the "final specification" has a nontrivial formulation. The designer must leap to the right conclusion in just one step of the development computation.

Clearly, we wish to develop a much richer model for development processes so that we may support it with a useful and automatable methodology. We cannot do this for an arbitrary process such as (D,S) above since it is not subject to any further formal constraints.

We will postulate that all development processes will be abstract discrete processes, and leave all nondiscrete transformations to be carried out during a process step. The state space of such processes will be specifications at various levels of abstraction. The successor relation will be informally defined by the designers actions in producing a successor specification. Clearly our abstract development processes will not formalize all aspects of system development. That is impossible. We can formalize some aspects without prejudicing the (informal) treatment of the remaining aspects. One measure of the power of our design theory will be the range of aspects that can be usefully formalized.

Our hypothetical development process can thus be described as being in a design state (that specifies the relevant design decisions) that is transformed by the designers into a new

design state that specifies the current design decisions. Thus a particular system design sequence is a computation of the development process.

The generation of a system design via such a computation in a single step is vastly too complicated. We will thus want to support development processes whose computations consist of many (much simpler) steps. We may also wish to define subsequences of a computation as phases. A phase of a development process is a member of a partition of the computations generated by that process. The types of design decisions may be quite dependent on the phase of the development process. Typical examples of development phases are requirements, process design, implementation, deployment, etc.

Each phase or computation step thus focuses on a more specialized system development problem (i.e., how do we get from this state to that state?). This is one way to partition a design problem into separate problems. Such design problem partitioning is essential if the designer is to succeed. There are other very important system development process decompositions that must be developed as part of an application design theory. This is a vital subject but it is outside of the scope of this metatheory discussion.

The phase type of development partitioning focuses on more detailed and simpler subproblems involving system design decisions. Unfortunately, this partitioning does not simplify

the design theory problems involved, since the theory of how to go from one state to some subsequent states is largely independent of which phase is involved. The complete solution to a development problem in one phase, no matter how narrowly constrained may require results from an entire methodology or design theory. There can be some theory specializations for a given phase but we must first find the more general theory before we define specializations. We must thus find a different and more useful partitioning into subproblems for our research process to develop the required design theory.

3.1.3 Design Theory Development

The research process required to generate a particular design theory could also be formalized but this is unnecessary for our purposes. In the absence of a satisfactory design theory, we will happily settle for a useful theory rather than a theory of such theories. We will thus treat the research process quite informally. Further, until we have a useful design theory, arguments about a better one are premature. We are not after an optimum theory (how could you even tell if you had one?) but rather a useful theory. Elaborations, evolution, and experience will then get us a better theory.

We will thus postulate a particular research "computation" that will result in a design theory. This "computation" may not be optimal, but it has the essential virtue of being practical and we can (and have) carried it out to produce a

first cut at a design theory. The justification of this postulated research "computation" is, of course, in the usefulness of the resulting theory. This approach is intuitively plausible and is far from a random selection. There seems little point in elaborating a discussion of how and why we created this particular approach. It can stand on its own results. A brief description of the overall research "computation" in the form of a research procedure is given in Figure 3.1-1.

The basic problem discussed in this section, is that of decomposing the research process so as to simplify its steps. We can illustrate the relationship between development process partitioning into phases and the research process decomposition described by this research approach in Figure 3.1-2. The source of our previous difficulty with phase decomposition is now obvious. No matter how small a development step we focus on, we need the results of all of our research process steps to support it. Thus such focusing does not simplify our research problem. Our approach first identifies and develops those aspects common to all steps, second focuses on special phase sensitive problems, and third develops the application sensitive specializations.

A more detailed discussion of Figure 3.1-1 is given below, and constitutes an informal description of our research process.

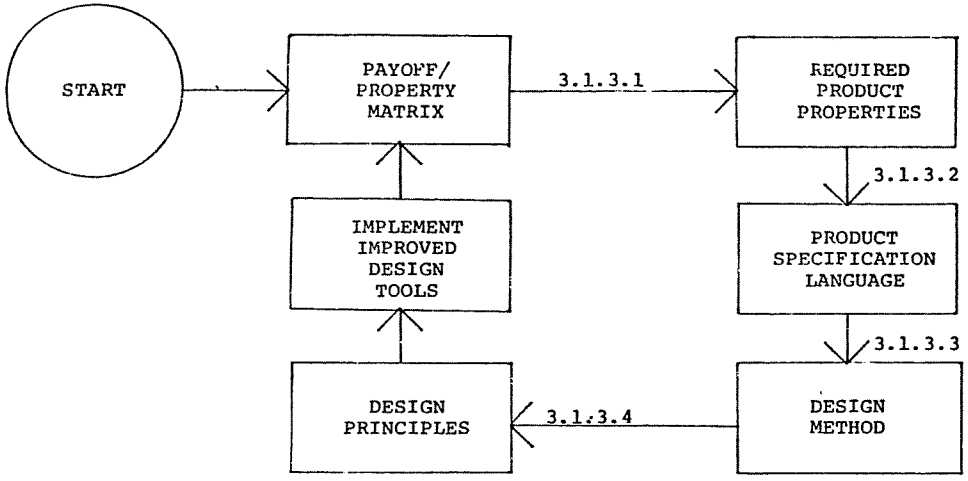


Figure 3.1-1 A Research Procedure

The section numbers on the arrows refer to the corresponding discussions.

3.1.3.1 Payoffs to Properties

We can start with a characterization of the payoffs to be attained or improved by the resulting design theory and a characterization of the most significant properties of the development processes supported by that design theory. These payoffs and properties must generate requirements on our design theory rather than on an application design to be produced using our design theory. We can then study the impact of those significant properties on the desired payoffs. We may be able to develop explicit functional relationships between them, in which case we can transform the payoffs into objective functions of development attributes and can quantify the "goodness" of the design.

For many, if not all properties, we will only be able to develop a qualitative (or semi-quantitative) impact matrix that identifies major dependencies. For example, rapid development may be a highly desirable payoff of a development process. We do not have a formal model of development speed that will permit a quantitative prediction based on development properties. It is important to note that dependencies that cannot be predicted in an application independent way (e.g., those that could only be obtained by monitoring a particular system development) will be of little use in developing a general theory. They may be used in the latter development of specializations of the design theory.

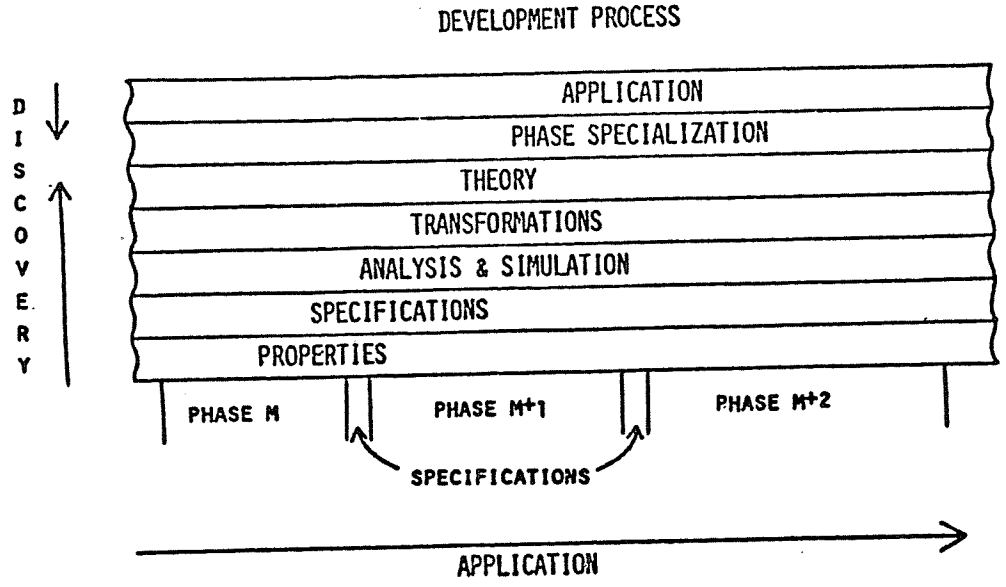


Figure 3.1-3 Research vs. Application Processes.

An application development will normally proceed from left to right through phases. A theoretical research development will flow from bottom to top. An empirical research development will flow from top to bottom by generalizing from application experience in a phase sensitive way.

This analysis then generates a set of critical properties that should be present for all development processes supported by the design theory we wish to develop. If we quit now, we would still have generated some useful results that could provide guidance to designers. However, we need not stop and we can easily proceed to the next step.

3.1.3.2 Properties to Specifications

Given the set of properties produced by the previous procedure, we can identify a subset of them that are properties of the system being developed. We now postulate that any allowable system specification must have these system properties or can be practically tested for them. This is the source of major requirements on our design of system specification languages. Clearly such properties are not sufficient to generate a unique specification language nor are they sufficient to completely characterize a specification language. They do represent necessary constraints on specifications.

A formal language is a set of strings over a vocabulary of characters generated by a formal grammar. A specification language is a subset of a formal language such that each sentence in the language can be practically tested for the necessary properties. A system specification is thus a sentence in a specification language that is interpretable to include the design decisions and to characterize the specified system.

Note that all specifications in a specification language must be practically testable for the necessary system properties.

It is unlikely that we will have thought of all such properties the first time around. Good judgment on the part of the researcher is just as essential as good judgment on the part of the designer using the resulting theory. Iteration of this research procedure will clearly be required. However, the resulting design theory may be quite sensitive to the ordering of such properties created by incremental iterations.

We must start with the most basic properties on which other properties are dependent. For many applications of this research approach, we may actually be able to establish a partial ordering on the properties such that the existence of a later property is dependent on the existence of the previous properties. This property dependency ordering would itself provide valuable guidance to designers.

We can see in advance the need for a specification to be suitable for the automatic generation of the behavior (computations) of the specified system. Without such feedback, the designers must be severely handicapped. They could not even test hypotheses about the effects of their design decisions. We will postulate the necessity of this property for our design theory.

If we quit now we would have developed a system specification language that will allow verification of the

presence of these necessary basic properties. We also have a precise way to specify current design states and design decisions as well as design problems. This alone can significantly improve our ability to formulate, communicate, and solve design problems. The required properties can be automatically tested. We can now go on to the next step, a methodology.

3.1.3.3 Specifications to Methodology

A methodology consists of a set of procedures for producing specifications, automated tools to support the procedures, and rules as to the applicability of each procedure and tool. An effective procedure is one that can be carried out even if it will never complete. An algorithmic procedure is an effective procedure that will terminate in a finite time. We will postulate that the procedures of our methodology must be effective at least. We cannot hope to make all such procedures algorithmic.

The range of the methodology procedure (viewed as implementations of relations) is a set of specifications. We can define a homogeneous methodology as a methodology whose procedures have domains that are a subset of their ranges. We can now define a homogeneous development process that is implemented by a homogeneous methodology as (D,S) where S is a system specification language and D is a set of predicates (defining the applicability rules) and relations (defining the methodology transformations). Such a development computation

starts with an initial system specification and finishes with a final system specification.

A homogeneous methodology is not very interesting if all specifications in the specification language have totally disjoint behaviors since there would be nothing in an initial system specification that is relevant to any subsequent system specification and thus it could just as well be bypassed. This repeated bypassing leads us to the trivial development process that we have already rejected as useless.

We will thus postulate that a specification language must have a specification for each member of some hierarchy of system abstractions for each final system specification in the specification language. The homogeneous development process can thus start with a very abstract system specification and end with a very detailed system specification. We now have an interesting development process model.

We can identify potential procedures for our methodology by working our development computations in reverse, as shown in Figure 3.1-2. In effect we can take a final system specification and ask "from what more abstract specification could we define a procedure that could get us this final specification?". Each such procedure is a potential candidate for our methodology. We can then iterate with the new starting point to find a more abstract starting point. One measure of the power of our methodology is the degree of abstraction

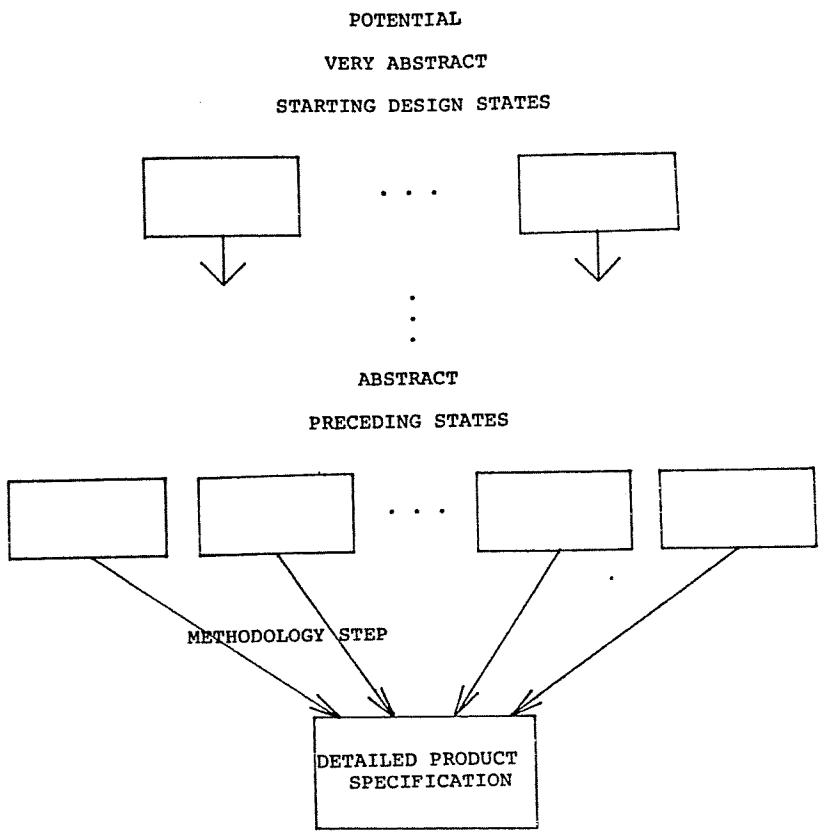


Figure 3.1-2 Design Processes

Each state is a "sentence" in a specification. A development "computation" will be some path from a particular starting design state to a final detailed product design.

of potential starting points for which there exists a development computation supported by that methodology.

A weak methodology might require us to start with some final system specification and thus provide no assistance to the designer. A strong methodology might start with a very abstract system specification at the system requirements level and end in a detailed final implementation specification.

Perhaps an admittedly trivial example will clarify this procedure. Suppose our specification language consisted of simple arithmetic expressions involving primitive sets and functions. Let F, G, and H be primitive functions and let X,Y be members of some primitive set of values. The expression F(G(X), H(Y)) has a valid abstraction in the form of P(X,Y) where P is a new primitive function. The procedure of elaboration (defining a primitive function in terms of other primitive) is a way to get from P(X,Y) to F(G(X), H(Y)) by defining P(X,Y) ≡ F(G(X), H(Y)). Thus a defining procedure for primitive functions that allows function composition is clearly a candidate for our methodology. When augmented by other forms of defining expressions, this procedure alone is sufficient to allow us to start from P(X,Y) and develop an arbitrary arithmetic expression with two arguments using a computation of the homogeneous development process implemented by our defining methodology.

The essential points of this methodology development procedure are that we need only study abstractions of a given specification language and that we can at least guarantee the existence of some development computation from any of the resulting set of abstract starting specifications. This does not guarantee that a designer will be able to discover a valid sequence. We must also develop some design principles to guide the designer.

We will postulate that there must exist a way to recognize that a procedure of the methodology has ended. We may place further constraints on such procedures but this one seems universally required. It is essential to demonstrate that a given computation step is finished before we can go on to the next one. The methodology procedure need not be algorithmic, since a designer may fail to complete the development step, fall back to some previous specification, and try another sequence of design decisions. We must at least know that it has produced a valid successor. Indeed it might have produced a valid successor that we cannot recognize. In that case we must assume that it has not ended.

This seemingly obvious postulate has some important ramifications and is a serious constraint on our methodology. For example, a typical development process step in current use is that of writing a computer program to implement some program specification. We do not have (currently) any way in general

to decide if the resulting program is a valid successor and does implement the specification. Thus this "normal" step is disallowed in our methodology except for specialized cases where consistency between the program and the specification can be shown. This rules out most of the current programming activities. If our methodology is successful, contemporary programming techniques will be radically altered.

Lest the reader become discouraged by this realization, let us hasten to point out that there are several viable alternatives. One of them is discussed below and uses heuristic procedures. A heuristic procedure is an effective procedure for generating a set of potentially desirable solutions using an algorithm for generating each potential solution. We can also define interactive heuristic procedures that allow some designer to guide the generation sequences so as to produce a better solution with less cost. If the interactive heuristic procedure is designed to generate all valid successor states in a development process, the designer can make random design choices and still generate only valid successor states while completing the computations step with any one of them. In its most automatic form, the procedure simply generates the "best" successor state first and we have an algorithm for our step in its least automatic form, we have an "idiot" procedure that requires intensive guidance by the designer to be practically useful.

3.1.3.4 Methodology to Design Principles

Our research process will produce development processes whose computations start with an abstract system specification and end with a detailed system specification. Each computation corresponds to some sequence of application of our methodology procedures. Many of these sequences may be nonterminating may require too many resources to carry out, or may result in a poor final design specification. Indeed, it can be assumed that a development computation is quite impractical unless it has been carefully selected. We clearly need more in our design theory.

Our methodology just concerns itself with carrying out particular computation steps. We must also develop some design principles to select suitable starting points and sequences of methodology application. We can usefully distinguish three kinds of design principles.

The first kind consist of absolute laws that delimit what is possible and result in a waste of effort if not obeyed. We have a rich source of such absolute design laws in the formal impossibility of most of our design tasks (unless we place serious constraints on both the developmental and the system processes). For each thing that cannot be done, we can derive a corresponding absolute law which says "don't do that thing" and accept the burden of accomplishing our design some other way. For example, we must not use an arbitrary

methodology procedure in a development process, since we cannot test its correctness.

The second kind of design principle is a sufficient condition which, if met, will guarantee that we are not violating one of the design laws. We must know that we are not violating a design law. In general we may not be able to determine that we are not. We can accept such sufficient conditions as additional design principles. They are neither absolute laws (that say you must do it this way) nor unique (there may be many such sufficient conditions). With experience we may find the more useful ones and use them until we find better ones. With analysis and research we may find less restrictive conditions or extremely useful special cases. For example, if we do not specify looping within a process step, the classically unsolvable "halting problem" is testable.

The third kind of design principles simply embody design experience. "We tried this in a similar case and it worked better than that." Our formal theory development will not discover many of these kinds of principles itself. It may, however, provide useful generalizations of such experience. It will also provide a formal way to embody such experience and transfer it to other designers or projects.

Having come this far we can use the resulting design theory to design and implement a design laboratory system for

interactive design. We can then use that laboratory to test and extend our design theory.

We can also iterate on our research approach and build more specialized design theories based on more specialized aspects of our development processes. This will then extend the power of our formal design theory to ensure essential properties.

3.1.4 Conclusions

The research approach described in Figure 3.1-1 will, if carried out, result in a design theory. It can be carried out. The "goodness" of the resulting design theory will be dependent on the skill and judgment of the researcher as well as on the complexity of the systems to be designed. Such goodness can be assessed by measuring the desired payoffs on application experiments. However even a poor theory will be better than no theory, and the last section points out how even a poor theory can evolve to a better one through research and experience.

The resulting design theory will be well defined, formal, transferable, and teachable. It will also provide a way to profit from design experience in a formal way. These are essential attributes for a viable design theory.

From experience, we can surmise that the last, application sensitive specializations should be made only by experienced application engineers and designers. Thus it may well be more

effective to teach our basic design theories to the engineers and leave it to them to finish the application specialization. This is also a practical consequence of our formalization of basic design theory.

There may be many other approaches to decomposing a research project to produce a design theory. Ours will work. How well it will work is up to the researcher. It is a suitable starting point for the development of a science of design. We have used this research approach to develop an initial theory of design for distributed data processing systems. It worked and opened up a large new domain of automatable design tools, and provided new insight into an emerging science of design.

3.2 Distributed Data Processing Systems

We now have an informal research plan for developing a design theory. Before we can specify the development process properties that we will use for this theory of design, we must look at the general application area of large scale real time control systems. We are particularly interested in the use of distributed data processing architectures in the design of such systems. We will compare some important development payoffs and some relevant distributed data processing characteristics via an impact matrix. We will then develop some initial critical issues (payoffs) for our DDP design theory.

3.2.1 Development Payoffs

The major areas in which the current state of the art imposes severe constraints in the development of large scale real time control systems are discussed below.

3.2.1.1 Life Cycle Costs

The life cycle costs for large scale systems are severely constraining and prohibit development of all but the most critical systems. Even many critical systems must be prohibited because cost and benefit tradeoff studies rule them out. We need to lower life cycle costs substantially if we are to usefully exploit the potential of such large scale control systems. An analysis of life cycle costs indicates that by far the biggest cost factors lie in nonhardware areas of specification, design,

implementation and "maintenance." This last factor is dominant and is clearly misnamed. Bits do not wear out. Instead, we could more accurately call this area correction, redevelopment, and evolution.

3.2.1.2 Mission Confidence

There are two aspects to mission confidence. The first is, "Will the developed system design, if properly implemented, perform the required mission?" The second is, "Will this system realization, if properly operated, perform the specific immediately required mission?" The answer to both questions requires an understanding of system flaws.

The existence of requirement, design, implementation, and realization flaws in any large system (even in most small systems) is usually taken for granted, in spite of the sometimes catastrophic results. Extensive operational experience after the last changes have been made is our most reliable guide and the severest test we can currently have. Unfortunately, these measures are after the fact and, because of the frequency of changes, equally frequently invalidated. We need to improve our ability to prevent, detect, adapt, and correct such flaws both during development and during operation. We also need models of the processes that are useful in deriving confidence measures.

3.2.1.3 Rapid Development

The development system itself is a large scale real time control system that must produce timely responses. The current development cycles are much too long. In many cases, the development cycle spans much of the period in which the requirements and technology can be forseen. For some systems, the development cycle can forseeably extend past all foreknowledge of mission requirements and equally forseeably, get into severe difficulties. We need to reduce substantially current system development times.

3.2.1.4 Evolution

The major factor that severely impacts our ability to achieve the previous payoffs is the adaptation of both the development and application systems to changes in requirements, technology, and state of the system. Each such change may require a new development cycle (hopefully small scale, inexpensive, reliable, and rapid) that could jeopardize the entire system development. Adaptability to change is the single factor most often cited as of major significance by systems engineers. With a greater adaptability to change, we can defer many sensitive issues until later in the development process. In the limit, we could quickly design and deploy a kernel system that could evolve to meet full mission requirements in operational environments with benefit of actual

experience in its use. This would enormously decrease the omniscience currently required of the designers, if they are to anticipate future changes in mission and technology.

We need to anticipate the need for change in both development and application systems and provide techniques for minimizing the impact, assessing the impact, and proving the correctness of the changes that are made. With improvements in this area we can substantially impact the other development payoffs. With such improvements life cycle costs will be reduced, mission confidence will be enhanced, and the development cycle will be speeded up.

3.2.1.5 Conclusions

From the above considerations we can easily conclude that we need to improve the following:

-- the predictability of development behavior

-- the efficiency of development systems

-- the reliability of development systems

-- the testability of application requirement, design, and realization behaviors

-- the reliability of application system requirement, design, and realization behaviors

-- the adaptability to change of both development and application systems

This list is still too large and unfocused for practical use as payoffs for our design theory. We cannot hope as a first (or possibly a last) try at creating a design theory

that will address all of the important factors impacting these payoffs. Further, we must keep in mind that the payoffs we are after, in order to start our research process, are those of the design theory, not of the application system. It is clear that any significant progress towards the above goals will have an important and positive impact on our starting payoffs of life cycle costs, mission confidence, rapid development, and evolution.

We will first restrict our attention to the requirements and design phases of development which seem to be at the heart of many of the significant issues affecting the payoffs above. We will postpone considerations of application realization and deployment issues until we have a satisfactory prototype design theory. Clearly an improvement in requirements and design theory can have a substantial impact on the latter payoffs associated with realizations and deployment. Experience has shown that it is easier and cheaper to get things correct than to detect errors later and repair their damage.

3.2.2 DDP Characteristics

We are particularly interested in developing a design theory that will help us exploit the potentialities of distributed data processing in attaining the system payoffs discussed above. We will look first at the intrinsic properties of distributed data processing.

3.2.2.1 Distribution

The obvious property is that DDP systems may be logically or physically distributed either over a complex of local systems or over geographically dispersed nodes of local systems. Computing power can be collocated with devices being served. This may improve response times, decrease long distance band widths, increase reliability through redundancy, and increase survivability through both redundancy and local autonomy of useful functions.

We are more concerned with the impact of distribution on our design theory. Clearly the most obvious factor is that our design theory must be applicable to asynchronously interacting systems in a useful way. In our theory, we want to consider such interactions explicitly.

3.2.2.2 Architectural Domain

The obvious property is that DDP systems can utilize an enormous set of different architectures. From a technology point of view almost any conceivable interconnections of general or specialized digital systems can be constructed. This increases the range of design choices in solving developmental problems, while blurring (or eliminating at the design level) the distinction between hardware and software. We can thus defer the technology sensitive engineering decisions to later development phases. From a design theory point of view,

the increased architectural domain increases the complexity of design decisions by relaxing otherwise severe constraints.

3.2.2.3 Problem Adaptability

Because of the wider range of architectures, we can transform the perennial problem of "how can we shoehorn our software into this system?" into "which architecture best fits our functions?". In some cases, the answer may be centralized systems. In any case we can profit from the increased design freedom. This has been demonstrated many times in small systems where the complexity issues have been manageable.

From a design theory point of view we must be able to model both static and dynamic properties of DDP architectures in order to forecast the suitability of mappings of particular applications onto particular DDP architectures. Given the modularity of DDP architectures, we can usually fit the problem onto several architectures. We must model the performance of each in order to determine their suitability and to select among them.

3.2.2.4 Modularity

The wide range of DDP architectures includes highly modular systems whose response to loads and module failures can be modelled and predicted. A particularly simple form of evolutionary development, even in the operational phase, is to

increase the number of such modules. Such modularity can also simplify the behavioral forecasting by constraining the conflicts between processes for physical resources. These conflicts enormously complicate all dynamic models.

From a design theory point of view, modularity is a major tool in resolving complexity issues. A poor modularization will produce even greater design complexity. Chaos does not really promote design freedom. The potential lifting of performance limits by DDP architectures must be exploited to give us simplicity and usable design freedom. Instead of N ! complexity (in worst cases) we can obtain N^{-1} complexity (in best cases). In practice such extremes would not normally be encountered, but our position on this complexity scale is of dominant concern.

3.2.2.5 Conclusions

We may now augment our list of significant development system payoffs by the potential DDP payoffs of design freedom and simplicity. The resulting list of desirable payoffs for a development system is

-- predictability

-- generality

-- reliability

-- simplicity

-- efficiency

-- testability of specifications

-- testability of realizations

The achievement of these payoffs can be significantly more complicated in a DDP design. Thus a serious need for a design theory in centralized data processing becomes an absolute necessity for distributed data processing. It is this need that guides our DDP design theory development. We are now ready to start our research process with the definition of some of the properties our specifications must have.

The primary goal of our design theory is to improve significantly our ability to achieve these payoffs in a practical development system for DDP applications.

3.3 Specification Properties

We start our design theory development by postulating the following properties of specifications, each of which must be practically testable on any specification. The presence of these properties is essential for subsequent development of our design theory. These properties do not include all properties that could be formalized. These can all be plausibly justified for any discrete system development and are essential for most other properties. They were selected because of their significance in making specification language design decisions.

All of the following properties are, of course, testable on any specification developed by this research approach. The ambitious nature of these requirements is tempered by the knowledge that we have developed a specification language meeting all of them. These properties are defined to be generally applicable to any potential discrete system specification. They can thus be used to characterize and compare different specification languages.

3.3.1 Formality

Because of our emphasis on very large systems, we must restrict ourselves to specifications and techniques that can be formally defined -- because automated tools and supports are our only hope for taming complexity beyond what a single human mind can handle. Although this means that human factors in

design will not be addressed directly, the potential impact on them is still considerable. We cannot force a customer to understand completely all of his requirements at the outset, for instance, but we can hope to provide him with useful feedback at early stages of design, and to facilitate graceful evolution when requirements do change.

A specification is formal if it is an abstraction (i.e., a thing representing only a certain set of properties, instead of its literal self) such that its represented properties can be specified precisely. The imposition of formality as a requirement on our specification language implies that specifications will be abstractions rather than realizations. The relevant properties of the abstraction are precisely specified and potentially susceptible to automated analysis and transformation. Abstractions also provide us a "transparency" for our design theory. We will incorporate only the abstract properties into our theory. The remaining properties are neither treated nor prejudiced by our design theory. Where our design theory does not help, it does not hinder. An abstract development of our design theory provides a maximally evolvable base for particular development system designs and realizations. Most of the important human factor issues can be deferred to the subsequent design of a particular development system.

Our specifications must be formal if we are to develop a formal design theory. We must be able to specify our problems and potential solutions precisely if we are to provide significant extensions to current methodologies. We must specify approxmations to the desired systems. There is a vast difference between an informal and a formal approximation. The former prevents most automated analysis while the latter can be designed to make such analysis possible. Formality is essentially equivalent to testability. Without testability, our design theory becomes only wishful prescriptions.

Our formal properties will be based on specifications, computations, systems, and the relations between them. We will require the following definitions.

A specification L is a finite string over the global alphabet A. The specification language $\mathbb{L}$ is a set of specifications.

A computation C is a finite sequence of states, where a state is a finite string over some global alphabet C. The computation space $\mathbb{C}$ is the set of all computations.

A system A is a recursive set of computations such that:

(a) if $\langle\alpha\rangle,\langle\beta\rangle,\langle\gamma\rangle,\langle\delta\rangle$ are computations, $\sigma_i,\sigma_j,\sigma_k$ are states, and $\langle\alpha,\sigma_i,\sigma_j,\beta\rangle,\langle\gamma,\sigma_i,\sigma_k,\delta\rangle$ are computations in S, then $\langle\alpha,\sigma_i,\sigma_k,\delta\rangle$ is also a computation in S;

(b) if $\langle\alpha\rangle$ is a computation in S, then there exists a finite, nonempty set of states σ_i such that $\langle\alpha,\sigma_i\rangle$

is in S, for all i.

The system space $\mathbb{S}$ is the set of all systems.

The interpreter $\mathbb{I}$ is a relation in $\mathbb{L} \times \mathbb{C}$.

For convenience we concatenate sequences as follows: If $\alpha = \langle\sigma_1,\sigma_2,\sigma_3\rangle$, then $\langle\alpha,\sigma_4\rangle = \langle\sigma_1,\sigma_2,\sigma_3,\sigma_4\rangle$. These definitions state that there is a specification language $\mathbb{L}$ whose semantics are sets of computations in $\mathbb{C}$ as defined by $\mathbb{I}$.

To represent the states of a system as strings over some alphabet causes no loss of generality, since the formally defined system is only a stand-in for the informally defined, realized one anyway. The formal system is a set of computations, the same as would be obtained by observing and formalizing all possible computations of the corresponding realized system.

Property (a) in the definition of a system says that the subsequent behavior of a system depends only on its present state, and not on the past. This is characteristic of digital systems, simply because information about the past cannot be used unless it is encoded in the present state.

Property (b) in the system definition says that the system is cyclic, i.e., does not halt (any computation records a finite observation, but every computation is an initial subsequence of a longer one). This causes no loss of generality simply because a system which is intended to halt can go into a "null" state whose only successor is another "null" state.

The purpose of defining systems this way is to be able to distinguish inconsistencies in the specification by cases where a state has no well-defined successor. Thus the concept of a "halting state" is an interpretation by the user. This definition also accommodates both systems with single initial states and systems in which every state is a possible initial state of a computation.

Thus our system definition is very general; it is hard to imagine any "digital system" which could have been left out. (The section on "asynchronism" will explain why it is adequate for distributed systems.) The reason that only state sequences appear explicitly in the definition, without mention of the processing between discrete states, is that this is enough for us to define the required "logical" or "functional" properties. Computation will have to appear explicitly in the system definition on the next iteration of the metamethod when performance properties will be added.

It is interesting to note that in the context of the metamethod, formality implies the decidability of $\mathbb{L}$, i.e., that there will always be an algorithm that decides whether a given string is a member of $\mathbb{L}$. This is because, if specifications are formally defined, then the "testability" for other properties required by the meta-method must take the form of decision algorithms. Each decision algorithm determines a decidable language over which it is known to be defined, and

the intersection of these domains is the specification language: obviously decidable itself. The formal property of specifications is thus assured by the definition of specifications and need not be represented by a testing algorithm.

There are a large number of automatable analyses possible because of this property alone. For example, if a specification is proffered, it can be "syntactically" checked by a "parsing" algorithm to decide that it is completely specified and correctly formed. Even this testing is not possible in some currently used specifications. We can provide to a designer the type of feedback from a "compiler" currently obtained by a programmer. The utility is obvious.

3.3.2 Consistency

We will also require that a specification does specify a system. This is a nontrivial property since it can easily be violated by specifications. For example, if the specification consisted of a set of equations whose solution specified a system, we would have to have an algorithm to decide whether or not there existed a solution. This is in general not possible, and requires very severe constraints on the nature of the equations.

For each L in $\mathbb{L}$, L is <u>consistent</u> if the set of all related computation sequences is a system in $\mathbb{S}$.

Consistency of a specification means that its image under interpretation is a system. It is a very important property because it precludes both "syntactic" errors (missing parts, double definitions, etc.) and "semantic" errors (infinite loops and deadlocks) which would prevent computation of a successor state to any state. Thus any consistent specification specifies validly some system.

This definition of consistency also subsumes the unambiguity of the specification of L. Each specification will specify a unique system. The property of consistency is possessed by few of the current forms of system specifications. A designer has an obvious interest in the possession of this property by system specifications.

3.3.3 Effectiveness

The consistency of a specification ensures the existence of a system meeting it. This does not imply that we can determine whether a computation is an element of a specified system or not. We are in a position similar to a programmer who gets his program specification through a compiler but does not know what the resulting program will do. The program behavior can be studied with trial computations. We also want the behavior of a specified system accessible and testable.

We will thus require the effectiveness property of our specifications.

The property of effectiveness means that a specification, regardless of how abstract it is, is "runnable," i.e., can be used as a simulation model of the specified system -- to the level of the properties that have been specified. This idea has been in circulation at least since Zurcher and Randell ([ZR]) recommended that a system evolve from simulations of itself. It is realized in the SREM project ([BB], [Al], [DV]), in which the "functional" or "analytic" properties of a requirements specification can be simulated by providing simulations of private processing functions with behavioral or performance attributes, respectively.

Effectiveness is of fundamental importance because it provides early feedback to the designer and his customer about the behavior of the specified system. In terms of our meta-method, it provides the only possible handle on those properties not chosen to be guaranteed by the design method: as soon as the specification is elaborated to a point where those properties are defined, they can be tested by any conventional means.

It seems that the most useful formulation of effectiveness would make it always possible to generate all the states which could follow a given state in a computation of the specified system. This correspond most closely to our idea of "running" a system, and, by clause (a) of the system definition, it is

then possible to construct any finite subset of the system.

A consistent specification L is effective if there exists an algorithm which, given any state σ appearing in a computation of $S = \{C:(L,C) \in \mathbb{L}\}$, will produce the set of all states σ' which are immediate successors to σ in computations of S.

Note that this definition cannot be satisfied unless the set of all successor states is finite. This is guaranteed if Σ, the set of all states appearing in computations of S, is finite. And finiteness of Σ is useful and practical in its own right -- all realized (and realizable!) digital systems use finite amounts of memory resources.

Informally, the possession of this property ensures that we can provide a universal (for all specifications in $\mathbb{L}$) procedure for evaluating a specification and generating instances of the computations of the specified system, i.e., a universal system simulator running directly on the specification itself. There can thus be no discrepancy between the specification and the "simulation" model.

The designer could thus obtain test computation data directly and automatically from the specification itself. Debugging design decisions are now possible, even with abstract specifications. A trivial way to obtain this property is to specify systems by programs that can be compiled and interpreted.

3.3.4 Homogeneity

The next desirable property of a specification is that every abstraction of the system it specifies have a specification in the same language. This is called "homogeneity" because it means that the results of all design phases can be expressed in the same homogeneous language: system architectures can develop from requirement specifications, and the (admittedly arbitrary) distinction between hardware and software disappears.

The necessity of this property is clear from the current wide interest in data abstractions, design languages, etc. It is simply impossible to expect people to design complex systems without mechanisms for expressing abstractions of them, or to choose particular abstractions for them.

We want our specification language to include both specifications of a given system and specifications of abstractions of that system. We can then use the same specification language from the start to the finish of each development process. This will eliminate the introduction of errors in going from one specification to a hopefully equivalent specification in the language of the next development phase. The entire methodology is then available in each phase.

There are two forms of abstractions which can be defined in the system domain, both of which will later be related to specific design steps.

A system S is a containing abstraction of a system S' if $S' \subseteq S$.

A system S is an embedded abstraction of a system S' if there exist projection functions p_f: $\langle A^* \rangle \rightarrow \langle A^* \rangle$ and p_s: $A^* \rightarrow A^*$ such that $\forall C \in S$, $\exists C' \in S'$ such that $C = p_s(p_f(C'))$. (p_f removes states from sequences, and p_s removes characters from states.)

A system S is an abstraction of a system S' if it is a containing abstraction or an embedded abstraction of it. How do these relationships arise? Let L be a specification with a primitive computation state successor function $f: \mathcal{D} \rightarrow R$ and let L' be the same specification, but with f elaborated in terms of lower level primitives. The system S specified by L contains computations in which f maps every value in $\mathcal{D}$ onto every value in R, while in the system S' specified by L' some of these computations have been eliminated by the additional structure in f. Thus S is a containing abstraction of S'.

On the other hand, let S be a system modeling execution of a program (its states are values of the vector of variables, and its steps are statement executions), and let S' be a system modeling the implementation of the programming language on a computer (its states are machine states, and its steps are instruction executions). Then S is an embedded abstraction of S', with p_f removing all states of S' that arise during

execution of language statements, and p_s removing all state information except the user-defined program variables.

A consistent specification L' is homogeneous if for every system S which is an abstraction of $S' = \{C':(L',C') \in \mathbb{\Pi}\}$, there exists an $L \in \mathbb{L}$ such that $S = \{C:(L,C) \in \mathbb{\Pi}\}$.

The computations of the less abstract system are thus contained in the computations of the more abstract system. This is essentially a concept of system generality. The less general system has a subset of the computations of the more general system. If the general system is satisfactory in each computation then the special system will also be satisfactory with respect to the same aspects.

Homogeneity implies that every abstraction of a specification is also a specification. As a result, any development process starting with an abstract specification and ending with a less abstract specification can have intermediate states that are also abstractions of the target specification. All of those states are specifiable. All properties of the abstract system are automatically inherited by the less abstract system produced from it. Further design decisions do not invalidate past design decisions provided only transformations to less abstract systems are used in development. This property also allows direct traceability of design decisions at each level.

3.3.5 Modularity

We must have some way to localize design decisions and control the complexity of the design. A modular specification is one with identifiable components which can be replaced by compatible components, producing only local and predictable changes in the specified system. Modularity is essential in the specification of complex systems, because it makes it possible for one person to understand parts of the specification [BH], and for many people to work on parts of a large design data base.

There may be many forms of modularity, but only one is sufficiently basic and language independent to be defined here. It is the concept that elaboration, or replacing a component by a less abstract one, creates a specification which is less abstract -- in the sense that the system specified by the former is a containing abstraction of the system specified by the latter. To formalize this, we must define "component" and "more abstract component."

A component K is a finite string over the global alphabet A. The component language $\mathbb{K}$ is a set of components.

The component relation $R_{\mathbb{K}} \subseteq \mathbb{L} \times \mathbb{K}$ contains a pair (L,K) if and only if K is a substring of L which is a component.

The abstraction relation $R_A \subseteq \mathbb{K} \times \mathbb{K}$ contains a pair (K,K') if and only if K is an abstraction of K'.

In a program, a component might be any substring generated from a nonterminal of the context-free grammar. A procedure whose body is undefined is an abstraction of a procedure of the same name and parameter list with a defined body.

A homogeneous specification L' is modular if for every K' sucn that $(L',K') \in R_{\mathbb{K}}$ and every K such that $(K,K') \in R_A$, the specification L, formed by substituting K for K' in L', is either inconsistent or such that $S = \{C:(L,C) \in \Pi\}$ is a containing abstraction of $S' = \{C':(L',C') \in \Pi\}$.

The reason that L can be inconsistent, even though K is a valid abstraction of K', is that consistency is intrinsically global. For instance, K might be a primitive function, and K' might be an elaborated version in which an interaction with another part of the system (perhaps to get control information) is specified. The substituting K into a consistent specification containing K' (which must have a specification of the other half of the interaction to be consistent) will create an inconsistent specification, in which the other half of the interaction is left hanging.

Informally, the modular property provides the ability to develop components with a process similar to that supported by homogeneity. In effect, modularity is equivalent to local homogeneity. We thus design components independently while preserving previous properties and design decisions of the more abstract (general) specifications.

The component relation $R_{\mathbb{K}}$ may also be applicable to components themselves. In this case we can define components in terms of components and establish a hierarchy of modularization. This will vastly improve our ability to factor the design complexity.

3.3.6 Informal Extensibility

A specification language needs to provide for comments and other information expressions of the designer's choice. The distinguishing characteristic of such informal expressions is that they do not affect formal interpretation of the specification. Thus the definition must distinguish between pairs of specifications which differ only in uninterpreted attributes, and pairs of specifications which specify the same system via different interpretations. This is done by associating uninterpreted attributes only with modular components. Informal attributes may often become formal ones during subsequent iterations of the metamethod.

An informal attribute set T is a finite string over the global alphabet A. The informal attribute set language $\mathbb{T}$ is a set of informal attribute sets.

The informal attribute relation $R_{\mathbb{T}} \subseteq \mathbb{K} \times \mathbb{T}$ contains a pair (K,T) if and only if T is a substring of K which is an informal attribute set.

A modular specification L is informally extensible if for every K such that $(L,K) \in R_{\mathbb{K}}$ and every T such that

$(K,T) \in R_{\mathbb{T}}$, the specification L', formed by substituting T' ($T' \in \mathbb{T}$, $T \neq T'$) for T in L, is such that $\{C:(L,C) \in \Pi\} = \{C':(L',C') \in \Pi\}$.

Because our formal specifications do not include all properties of interest, we must provide some way to include uninterpreted (informal) attributes that convey the desired information. Our methodology will not analyze such attributes since they are not formally expressed. However, any informal methodology may be used with respect to these uninterpreted attributes. We won't provide much assistance other than that of a controlled data base, but we won't hinder that which can be done by the designer.

The extensible property provides an "open end" where properties we do not yet wish to formalize may be included informally in our specification. A component seems to be a very natural "unit" to possess such attributes.

3.3.7 Distributed

A specification must be able to define distributed systems if it is to address the essential DDP design problems. A formal definition of the "distributed" property is developed below. Distributed systems differ primarily in that the system computations are composed of asynchronously interacting computations of distributed system components. Distribution is also important for decomposing complexity in a nondistributed

system. At a low level of abstraction, most systems are distributed.

3.3.7.1 Asynchronous Subsystem Compositions

Many formulations of asynchronous interactions have been proposed, but what we need here is a definition of this property which is independent of the mechanism or its implementation. The essence of "being composed of asynchronous processes" seems to be that the specification can be factored into separately interpretable specifications, and that the aggregate computations of the system are composed from computations of these parts, taken at all possible relative rate combinations. The essence of interaction between these "processes" seems to be to constrain the computations just described. The information received by a process in an interaction serves to rule out some otherwise possible computations, just as the information gained by elaborating a primitive function rules out some mappings from elements of its domain to elements of its range.

Let $L_1, L_2, \ldots, L_n$ be effective specifications, and let G_i be the relation on states which maps a state of the system S_i specified by L_i to its finite set of successor states, $1 \leq i \leq n$. Then the asynchronous combination of $S_1, S_2, \ldots, S_n$ is the system:

(a) whose states are n-tuples $(\sigma_1, \sigma_2, \ldots, \sigma_n)$, σ_i a state in the computations of S_i, $1 \leq i \leq n$;

(b) whose successor relation G is defined by:

$((\sigma_1, \sigma_2, \ldots, \sigma_n), (\sigma_1', \sigma_2', \ldots, \sigma_n')) \in G$ if and only if:

(1) a unique i, $1 \leq i \leq n$, $\sigma_i' \in G_i(\sigma_i)$;

(2) $\sigma_j = \sigma_j'$, $\forall j \neq i$.

Then the definition of asynchronism states that the asynchronous combination of the process interpretations is a containing abstraction of the specified system with interacting processes.

In effect, this notion of composition produces a new system whose joint computation sequences correspond to all combinations of computation steps by the n subsystems. The subsystem sequences are preserved by embedding them in the composed system sequences.

The constraints to unique values of i is equivalent to requiring that the "events" of computation step completions never occur simultaneously, i.e., there is no true simultaneity between asynchronous computations. This constraint is very useful, and we can always model the "simultaneous" systems by asynchronous sytems.

3.3.7.2 Asynchronous System Specification

An effective specification L is asynchronous if there is a function which maps L to a finite, nonempty set of effective specifications $L_1, L_2, \ldots, L_n$, such that the asynchronous combination of $L_1, L_2, \ldots, L_n$ is a containing abstraction of $\{C: (L,C) \in \mathbb{I}\}$.

The asynchronous system is thus contained in the corresponding asynchronous combination. The effect of an interaction between subsystems of the asynchronous system will be to eliminate certain computations from the corresponding asynchronous combination. If no interactions occur, the asynchronous system is the same as its asynchronous combination. By defining asynchronism without defining interactions, we avoid making any restrictions that might exclude distributed systems. Even the most general discussion of distributed systems ([La2]) acknowledge that a distributed system has a well-defined global state; it is just that in a distributed system, knowledge of the global state on which to base decisions is harder to come by.

A fixed process structure seems to be the inevitable result of reasonable definitions and manipulations of asynchronous systems. There are other sound arguments for static process structures, especially since dynamic reconfiguration can be built in as an evolution (see also [BH]). Multiprogramming systems, for instance, usually have I/O processes and user processes. But the I/O processes correspond to a fixed configuration of devices, and the degree of user multiprogramming is fixed or bounded.

We can thus design and study the properties of the subsystems in isolation, knowing that their integration will not produce new and unexpected behavior. Subsetting of the

computations is all that can occur. This property of distributed systems will be quite important for any development process in which subsystem integration is attempted.

3.3.8 Generality

A final interesting property of a specification language is completeness: having at least one specification for every system.

A specification language $\mathbb{L}$ is complete if for every $S \in \mathbb{S}$, there exists an $L \in \mathbb{L}$ such that $\{C:(L,C) \in \Pi\} = S$.

This is not a product property at all, but rather a property of a design process. It seems that completeness is a theorem to be proved as a part of the design principles, especially since completeness may be deliberately compromised. For instance, as hinted in the section on effectiveness, we have no intention of allowing the specification of systems with infinite state spaces.

3.3.9 Conclusions

We have defined an abstract set of properties for specifications and specification languages that are critical for creating a formal design theory that will enable the achievement of the DDP payoffs.

A suitable specification language for distributed systems must have at least these critical properties in a useful form. We may use these properties as requirements for specification language design.

We will clearly add further properties as required to support subsequent development of our design theory. These will be introduced in latter sections.

Figure 3.3-1 is a precedence graph of the required product properties. P_1 precedes P_2 if the definition of P_1 is needed to define P_2, and "L has P_2" implies "L has P_1."

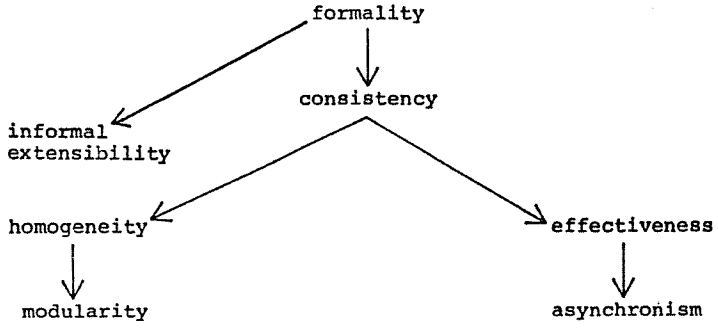


Figure 3.3-1: The required product properties.

The property definitions, in addition to their usefulness in our application of the metamethod, are interesting in their own right. Being formal yet independent of any specification language, they can serve as verifiable requirements on the

design of a specification language. For instance, in [Al] the following desirable properties of a specification are named: "completeness, consistency, correctness, testability, unambiguity, design freedom, traceability, communicability, modularity, and automatability." Alford goes on to say:

> As a result of the foregoing analyses, three goals were then identified for an SREM: (1) a structured medium or language for the statement of requirements, addressing the properties of unambiguity, design freedom, testability, modularity, and communicability; (2) an integrated set of computer-aided tools to assure consistency, completeness, automatability, correctness; and (3) a structured approach for developing the requirements in this language, and for validating them using the tools.

Using our results, these requirements for a specification language and its associated tools can be evaluated as follows.

Probably everyone would agree that although "communicability" is a desirable goal, it is intrinsically subjective, and thus subject only to personal evaluation.

Our definitions put "automatability," "modularity," and "testability" into precise terms as the properties of formality, modularity, and effectiveness, respectively. The importance of this should be clear from the vagueness of a term like "modularity" without a formal definition. "Modularity is enhanced by the maintenance of the requirements . . . in a centralized data base . . ." ([Al]) is simply not enough to determine whether a given specification language has "modularity," or whether such "modularity" is really worth

having. To the extent that "design freedom" means naturalness to a human designer, it falls in the same subjective category as "communicability"; to the extent that it means that every system can be designed, it is defined precisely in the next section as completeness. In either case, the text is not sufficient to distinguish which was meant.

The other four properties addressed are "consistency," "completeness," "unambiguity," and "correctness," with decision algorithms for "completeness" and "consistency" being mentioned explicitly. Our analysis shows that in all formal senses, these are one and the same property: consistency. A consistent specification specifies, under a formal interpretation, a valid system (although that system may be very abstract, i.e., unelaborated). The specification must be internally consistent and "complete" (have no parts missing); it then unambiguously and correctly specifies that system. The system may not be what the user wanted, but this is not subject to automated verification.

The most significant omission from these specification requirements is that of performance properties. Our research approach is first to develop a design theory of what a distributed system does and only then to address questions of how well it operates. We can provide substantial payoffs even prior to considerations of performance. The extensibility property required of our specifications may be used to include performance requirements in our formal specifications. Our

initial design theory may assist (through its formal analyzability) performance analysis, and will certainly not prevent a designer from using any otherwise feasible performance methodology.

The required specification properties are sufficient to resolve most of the difficult design decisions involved in producing a DDP specification language.

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

[ZR] Zurcher, F.W., and Randell, B. "Iterative Multi-level Modelling — A Methodology for Computer System Design." Information Processing 68, A.J.H. Morrell, ed., North-Holland, 1969.

[BB] Bell, Thomas E., Bixler, David C., and Dyer, Margaret. "An Extendable Approach to Computer-Aided Software Requirements Engineering." Trans. Soft. Eng. SE-3, January 1977.

[Al] Alford, Mack W. "A Requirements Engineering Methodology for Real-Time Processing Requirements." Trans. Soft. Eng. SE-3, January 1977.

[DV] Davis, Carl G., and Vick, Charles R. "The Software Development System." Trans. Soft. Eng. SE-3, January 1977.

[La2] Lamport, Leslie, "Time, Clocks, and the Ordering of Events in a Distributed System." Massachusetts Computer Associates, Inc., March 1976.

3.4 Specifications

We will now describe, both formally and informally, a specification language that will possess the required properties developed in the previous section. This specification language should be considered as a base line design that is subject to later revision as required in the development of our DDP design theory.

We will first introduce an algebraic characterization of the specification language concepts and their interpretation as systems. The next section deals with the simpler case of non-interacting systems (i.e., nondistributed systems) and provides a basis for the following discussion of distributed systems.

The specification language itself is a "source language" that is translated to a semantic data base structure. This semantic data base is the form on which analysis and transformations will be applied.

The semantic data base is incorporated in a Computer Assisted Formal Engineering Laboratory (CAFEL) that will support the use of our DDP design theory in design experiments.

Finally, we will describe some of the important analysis tools that can be included in the CAFEL.

3.4.1 Nondistributed Systems

A specification language $\mathbb{L}$ has the semantics $\mathbb{C}$ defined by $\mathbb{I} \subseteq \mathbb{L} \times \mathbb{C}$ where $\mathbb{C}$ is a set of discrete computations, and $\mathbb{I}$ is an "interpreter" for $\mathbb{L}$. We will develop our specification language in the form of algebraic definitions of generating relations for elements of $\mathbb{C}$. These relations will be defined as processes.

A process is a pair (f,Σ), Σ a state space, $f: \Sigma \to \Sigma$ a successor relation. A computation of a process is a finite looping sequence of states $\sigma_0,\sigma_1,\ldots,\sigma_j,\ldots,\sigma_j$ where $\sigma_i \in \Sigma$ and $\sigma_{i+1} = f(\sigma_i)$, $i \geq 0$. A state space is defined in terms of primitive sets. The successor relation is a (possibly nondeterministic) function defined in terms of primitive functions. Neither primitive sets nor primitive functions are formally defined in a process. They may, of course, be defined or described in extensions (i.e., comments) in our formal specifications.

Sets may be defined as primitive, as a set of literal values, or as a set valued expression involving union and products. Primitive sets will be used to define types. Literal values will be used to define constants. Set union provides multi-type definitions. The set product $N \times M \to \{(n,m) | n \in N, m \in M\}$ will be used to define structure in the state elements. These forms are sufficient to give us enough generality for our specifications.

Functions may be defined as primitive, or as expressions in terms of other functions. The forms of expression include composition, tuple formation, and selection. Primitive functions will be used to control the level of the abstraction of the specification (i.e., are they bit functions or do they operate on complex states?). Composition of function invocations will be used to generate ordinary algebraic expressions. Tuple formation will be used to define functions whose value is that of a tuple of expressions. Selection of expressions will be used as a conditional form.

Surprisingly, we now have sufficient algebraic structure to specify any nondistributed system in a form that has the required properties. The key to this simplicity is our definition of process and the insistence that a specification of a nondistributed system be a process. For designers used to thinking in programming terms, the process concept represents a radical change. If they will bear with us, we will later return to the notion of program and the relationship between processes and programs will be made clear. For now, a program may be considered as one way to implement a process. The advantage of processes over programs is that we can achieve a greater abstraction and generality using processes. Details of data representation and control need not be specified unless desired. The remarkable results of the homogeneity property is that, if such details are desired, they may also

be specified in processes. This will be illustrated in later sections.

3.4.2 Distributed Systems

We must be able to specify process interactions, both to partition a process into interacting processes for simplicity, and to specify distributed systems. We require a "functional" abstraction of interactions for compatibility with our algebraic definition of processes. The interactions between processes must be defined via primitive "functions" used in the definition of the processes.

3.4.2.1 Exchange Functions

A paper on the "Specification of Asynchronous Interactions Using Primitive Functions" is included as Appendix A. The details of the concepts of exchange functions are developed in that paper. A brief summary is presented here.

The first exchange function primitive is called XC_i, where i denotes its class. Interactions between exchange functions occur only within the same class. The evaluation of $XC_i(Z)$ may start when its argument, Z, is supplied but cannot terminate until a matching evaluation of another exchange function in the same class is selected. The selection mechanism is unspecified but is required to be fair (i.e., not exclude any in class from matching if enough exchange functions in

the class are evaluated). The matching requirement provides synchronization at some point during evaluation prior to completion. A member of a matched pair of exchange functions then completes its evaluation by taking the argument value of the other as its value. This "exchange" of argument values provides message transmission. Exchange functions are exactly like other computable functions in that their evaluation is initiated with an argument, and sometime later evaluation terminates, returning a value. Unlike many functions, however, exchanges may be nondeterministic, their evaluation has side effects, and they may fail to return a value if never matched.

We must also describe an exchange function XA_i. The only difference from XC_i is that two XA_i functions can never match each other. Thus any evaluation of an exchange function in a class containing only XA_i will never complete. The purpose of this primitive is to allow competition for matching an XC_i without allowing a match with one of the other competitors.

The final exchange primitive is XS_i. The only difference from XC_i is that an XS_i will always "match" itself if no other match is available when it is initiated. When XS matches itself, its value is its own argument and no other synchronization is required.

These three kinds of exchange functions are sufficient to specify very general message and synchronization facilities. As functions, they do not restrict their implementation to any particular form of interaction. Indeed, if matching exchanges are both implemented in the same storage no messages need be passed in the implementation. If matching exchanges are implemented in the same sequential procedure the match synchronization can be built in as a sequence of temporary assignments, and there are no explicit exchange operations. This could be the case if "cooperating sequential processes" were generated. More importantly, the exchange functions may be used to specify interactions at a high level of abstraction, independent of such details.

3.4.2.2 Tidal Wave Model

We will include an example of distributed system specifications taken from a system of differential equations. This demonstrates that our specifications are suitable for modeling "real world" continuous processes as discrete processes. This is the property of closure -- there need be no arbitrary and formally unspecifiable interface concept in our specification language. This has important consequences for our design theory.

This section describes a formal, high-level specification of a fluid simulation model by means of the formalism summarized in Appendix A of [Fi77]. The simulation model itself was designed and implemented previously by Greenspan et al. [GrCC76], although no acquaintance with the latter reference is necessary here. The purpose of the model is to study the propagation of tidal waves, rapidly moving shock waves generated by earthquakes at the ocean floor. The damage these waves can cause to possibly distant shorelines depends to a large extent upon the geometry of the ocean bottom immediately adjacent to the shore. Many of the details of the simulation model are omitted here since our description is primarily pedagogical in nature. However, a brief outline of the salient features of the model provides the foundation for an understanding of the somewhat abstract material to follow.

The simulation model is actually quite simple in design, even though it details with several complex natural phenomena. In brief, it models "particles," each of which represents a large quantity of sea water. These particles are contained in a two-dimensional ocean basin with a special region on one side which approximates the continental shelf and shoreline. The particles are held in the ocean basin by gravity and interparticle attraction; in addition, repulsive forces exist between particles (within a certain interaction distance) and between particles and the ocean bottom. A typical simulation

begins by assigning the initial position and velocity to each of a set of several hundred particles. Then the particles are allowed to move during enough discrete time steps to reach a stable configuration. Next, the geometry of a part of the ocean bottom is altered slightly but suddenly over the interval of a few time steps in order to simulate an earth tremor. Finally, the effect of this perturbation of the ocean floor upon the motion of the particles is observed for as many time steps as desired. Note that the forces between interacting particles remain constant throughout the simulation and that the essential shape of the ocean remains intact. However, these same factors may be modified in other simulations in order to find a more accurate model of tidal waves. In other words, the model is parameterized.

Our functional specification of the simulation model just described will assign a separate asynchronous process to each particle being simulated; each system step of a process then will represent a time step in the simulation. Note that even though in the simulation model we define particles to move discretely and synchronously, we have chosen asynchronous processes for our specification. There is really no contradiction here, as we shall see, since the processes are in fact loosely synchronized by messages exchanged among each other. Specifically, no two processes can be modelling time steps which are more than one simulated time unit apart at any given time.

We will formally specify this sytem as follows. Let n denote the number of particles in the model. Let P denote the set of processes which model the n particles:

$$P = \{P_1, P_2, \ldots, P_n\} \tag{1}$$

Each process P_i has two components. The first component, the state space Σ, is common to each P_i. The second component, F_i, is the state successor function for the process P_i.

$$P_i = (\Sigma, F_i), \qquad 1 \leq i \leq n \tag{2}$$

$$F_i: \ \Sigma \to \Sigma, \qquad 1 \leq i \leq n \tag{3}$$

We will elaborate the state space Σ by expressing Σ in terms of a position space X and a velocity space V as follows:

$$\Sigma = X \times V \tag{4}$$

Let $\sigma_i \in \Sigma$ denote the state space component of process P_i and let $x_i \in X$ and $v_i \in V$ denote the position and velocity of particle i. Thus:

$$\sigma_i = (X_i, V_i), \qquad 1 \leq i \leq n \tag{5}$$

We can decompose each state successor F_i into functions $F_{i1}: \ \Sigma \to X$ and $F_{i2}: \ \Sigma \to V$. F_{i1} is the position function of particle i and F_{i2} is the velocity function of

particle i:

$$F_i(\sigma_i) = (F_{i1}(\sigma_i), F_{i2}(\sigma_i)) \tag{6}$$

The position and velocity of a particle at time step $t+1$ depend on its velocity and position at time step t and on the position of other particles at time step t. We can elaborate the functions F_{i1} and F_{i2} to indicate these dependencies as follows: Let Δt denote the constant simulated time interval, m the mass of a particle, and $\Phi_i(x_i)$ the forces exerted on particle i. Then we have:

$$F_{i1}(\sigma_i) = x_i + (\Delta t) F_{i2}(\sigma_i) \tag{7}$$

$$F_{i2}(\sigma_i) = v_i + \left(\frac{\Delta t}{m}\right) \Phi_i(x_i) \tag{8}$$

The function Φ_i is defined by the following equation.

$$\Phi_i(x_i) = mg + B(x_i) + \sum_{k=1}^{i-1} R(x_i, XC_{ki}(x_i)) + \sum_{k=i+1}^{n} R(x_i, XC_{ik}(x_i)). \tag{9}$$

Here g is the acceleration of gravity and B is the time-dependent function which represents the force exerted by the ocean bottom on the particles above. The function R is the potential function which models the attractive and repulsive forces between two particles, the coordinates of which are its arguments. The second argument of each occurrence of R in Eq. 9 above is an exchange function which exchanges the coordinates of pairs of particles so that interactive forces

can be calculated. Note that the summations are constructed so that exchange functions are named so each particle exchanges with every other particle exactly once at every time step. (For any subscripts i and j, $1 \leq i < j \leq n$, XC_{ij} appears in the summation but XC_{ji} does not.) However, no particular order of summation or order of evaluation of summands is implied by Eq. 9; such details are not important at this level of specification. In particular either a serial or a parallel computation of Eq. 9 may be used in implementation.

We could at this point consider the functions B and R to be primitives. We would then have a complete high-level specification of the simulation model. However, we will elaborate the function R in order to illustrate our top-down design methods. First we should note that an optimization for our model is possible since particles interact only locally. This observation was exploited in [Gr77] to yield a computational complexity of $O(n)$ in the number of particles as opposed to conventional exchaustive interaction schemes of order $O(n^2)$. (Our computational complexity results apply only to serial computation. On the other hand, our functional specifications are in no way biased toward or limited to serial computation; computational details becomes relevant at a lower level of functional elaboration, as we have mentioned.) Moreover, we can indicate our optimization at a high level so that it could be implemented automatically at a lower level.

We simply define the function R so that the ocean becomes segmented into vertical zones, where particles may interact only within a zone or between adjacent zones. Let Z be a zone classification function. That is, the value of Z(u) is the zone for a particle with coordinate u. Using a Lisp-like selector function (see [Fi77]) we can define R in terms of a function R' as follows:

$$\begin{aligned} R(u,v) = (&Z(u) = Z(v) : R;(u,v), \\ &Z(u) = Z(v) - 1 : R'(u,v), \\ &Z(u) = Z(v) + 1 : R'(u,v), \\ &0) \end{aligned} \tag{10}$$

The function R' can be expressed in terms of a primitive distance function Dist and another function R".

$$R'(u,v) = (Dist(u,v) \leq d : R''(u,v),\ 0) \tag{11}$$

This equation indicates that particles can interact only at distances no greater than d. Finally, we define R" by the formula

$$R''(u,v) = c\left[\frac{1}{(Dist(u,v))^4} - \frac{1}{(Dist(u,v))^2}\right] \tag{12}$$

where c is a constant.

Several important conclusions can be drawn from the simulation specification technique which we have used in this section. First, the method of discrete simulation is perfectly

suited to the modelling of continuous real-world processes to any desired degree of accuracy. (Accuracy in this model is determined by the precision of the numerical computations and by the time step Δt, both parameters of the model.) By extension, we can model discrete real-world processes or hybrid combinations of both continuous and discrete processes, for example, a hardware processor with both digital and analog components. Second, we have a specification in such detail that implementation amounts simply to performing the calculations implicit in the equations. In contrast, if we had given a set of differential equations then the implementation would involve choosing methods for solving these equations, where not all methods would produce identical results. Furthermore, accuracy of the model would not be specified as a direct parameter but would nevertheless need to be considered in the choice of computation method.

[Fi77] Fitzwater, D. R., "The Formal Design and Analysis of Distributed Data-Processing Systems," University of Wisconsin Computer Sciences Department, Report CSTR 295, March 1977.

[CrCC76] Greenspan, D., M. Cranmer and J. Collier, "A Particle Model of Ocean Waves Generated by Earthquakes," University of Wisconsin Computer Sciences Department Technical Report 277, September 1976.

[Gr77] D. Greenspan, unpublished results.

3.4.3 Specification Language

The specification language is essentially a subset of ordinary algebraic notation with the introduction of the primitive exchange "functions" to define interacting discrete processes. We suppose that the designer of a specification may wish to use a language that is specially adapted to a specific application area, thus our formal engineering laboratory should be multi-lingual and accept a variety of specification forms. This may easily be arranged by defining our specifications as a structured data object that serves as an intermediate form for our specifications. Any desired source language that can be translated to our intermediate form is thus supported. This intermediate form is not very suitable for humans as a source language, so we must introduce a prototype source language.

3.4.3.1 Informal Specification Language

The normal algebraic specification of distributed processes is already a convenient notation that can be used informally (with usual conventions) to develop and communicate specifications. We will occasionally present examples in just that way. For example, the tidal wave model of the previous section is defined in that way. This is a familiar and natural notation to use. The new aspect is the interpretation of functions so defined as specifying systems. The caveat is that some algebraic forms are too complex to be used in our specifications and must be avoided. The next section will define the permissible constructs.

3.4.3.2 Specification Source Language

In this section we will provide a formal grammar and an informal semantics for a specification source language to be used by the systems designer. It may easily be mapped onto a closely related language designed for internal coding in conjunction with data bases and automated analyses. This latter language is described in the next section and those following it prefixed by 3.4.3.3. The correspondence between the two languages is so close that direct semantic parallels will be obvious to the reader and need not be pointed out. For the formal description of the source language we will use a notation recommended by Wirth and described in Appendix C of this report. The production rules are now numbered and listed without intervening text. A detailed description of their semantics will then follow.

(1) DEFINITION = BLOCK|SPECIFICATION|SET|FUNCTION.

(2) BLOCK = `LET´ NAME `BE´ DEFN_LIST `EB´ [ATTRIBUTES].

(3) DEFN_LIST = (DEFINITION|REPEAT DEFN_LIST`)´){`,´DEFN_LIST}.

(4) REPEAT = SYMBOL SUBSCRIPT `(´ SUBSCRIPT `,´.

(5) NAME = SYMBOL [SUBSCRIPTS].

(6) SUBSCRIPTS = (SUBSCRIPT|SYMBOL SUBSCRIPT`(´ SUBSCRIPT`,´
SUBSCRIPTS `)´]{`,´SUBSCRIPTS}.

(7) SYMBOL = {ALPHABET|DIGIT}ALPHABET{ALPHABET|DIGIT}.

(8) SUBSCRIPT = SUB_CHAR{SUB_CHAR}.

(9) SUB_CHAR = SUB_ALPHABET|SUB_DIGIT.

(10) DIGIT = `0´|`1´|`2´|`3´|`4´|`5´|`6´|`7´|`8´|`9´.

(11) SUB_DIGIT = `$_0$´|`$_1$´|`$_2$´|`$_3$´|`$_4$´|`$_5$´|`$_6$´|`$_7$´|`$_8$´|`$_9$´.

(12) ATTRIBUTES = `WITH´ STRING {`,´STRING}.

(13) STRING = ``´{STRING_ELMT}`´´.

(14) STRING_ELMT = DELIMITER|NON_DELIMITER|STRING.

(15) DELIMITER = `(´|`)´|`[´|`]´|`{´|`}´|`:´|`+´|`,´|`.´|
`0´|`≡´|`∪´|`×´ .

(16) NON_DELIMITER = DIGIT|SUB_DIGIT|ALPHABET|SUB_ALPHABET.

(17) LITERAL = STRING|BOOLEAN|INTEGER.

(18) BOOLEAN = `TRUE´|`FALSE´.

(19) INTEGER = DIGIT{DIGIT}.

(20) SPECIFICATION = `LET´ NAME `≡´ `(´FUNCT_LIST`)´[ATTRIBUTES].

(21) FUNCT_LIST = (FUNCTION|REPEAT FUNCT_LIST`)´){`,´FUNCT_LIST}.

```
(22)  SET = `LET´ NAME `≡´ [SET_EXP][ATTRIBUTES].

(23)  SET_EXP = SET_FORMER|SET_UNION|SET_CROSS|SET_INVOC.

(24)  SET_FORMER = `{´ITEM_LIST`}´.

(25)  ITEM_LIST = (ITEM|REPEAT ITEM_LIST`)´){`,´ITEM_LIST}.

(26)  ITEM = NAME|LITERAL.

(27)  SET_UNION = (SET_TERM|SYMBOL SUBSCRIPT`(´SUBSCRIPT`∪´
                  SET_UNION`)´)  {`∪´SET UNION}.

(28)  SET_TERM = SET_FORMER|SET_INVOC|`(´SET_EXP`)´.

(29)  SET_INVOC = NAME.

(30)  SET_CROSS = (SET_TERM|SYMBOL SUBSCRIPT`(´SUBSCRIPT`×´
                  SET_CROSS`)´   {`×´SET_CROSS}

(31)  FUNCTION = `LET´ NAME `:´ SET_EXP `→´ SET_EXP
      [`WHERE´ NAME[PARAMETERS] `≡´ FUNCT_EXP][ATTRIBUTES].

(32)  PARAMETERS = `(´PAR_LIST`)´.

(33)  PAR_LIST = (PAR_ELMT|REPEAT PAR_LIST`)´){`,´PAR_LIST}.

(34)  PAR_ELMT = [NAME]|PARAMETERS.

(35)  FUNCT_EXP = PAR_INVOC|FTUPLE|SELECTOR|FUNCT_INVOC.

(36)  PAR_INVOC = NAME.
```



```
(37)  FTUPLE = `(´EXP_LIST`)´.

(38)  EXP_LIST = (FUNCT EXP|REPEAT EXP_LIST`)´){`,´EXP_LIST}.

(39)  SELECTOR = `[´[PAIR_LIST`,´]FUNCT_EXP`]´.

(40)  PAIR_LIST = (PAIR|REPEAT PAIR_LIST`)´){`,´PAIR_LIST}.

(41)  PAIR = FUNCT_EXP `:´ FUNCT_EXP.

(42)  FUNCT_INVOC = LITERAL|NAME_LIST`(´[EXP_LIST]`)´.

(43)  NAME_LIST = (NAME|SYMBOL SUBSCRIPT
      `(´SUBSCRIPT `∘´ NAME_LIST`)´) {`∘´ NAME_LIST}.
```

The following semantic descriptions of the productions above are numbered according to the production(s) to which they refer. (Where productions are not discussed individually below, their purpose is considered obvious from related productions and mnemonic nonterminal names.)

(1) Every sentence in the source language consists of a definition, which in nontrivial cases is also a block.

(2) A block is a named sequence of definitions. Each definition in turn consists of a name plus the set, function, specification, or block associated with the name (see production (19)). Since blocks may be nested we adopt the usual convention that a name has scope only within

the block in which its associated definition occurs. Further, if multiple definitions occur for the same name in different levels of nested blocks then the innermost definition supersedes all similarly named definitions within which it is nested. Finally, multiple definitions within a single definition list, that is, where neither of two definitions with the same name is nested within the other, are not permitted for reasons of consistency.

(4) A symbol used as an iterator (dummy) variable has scope only within the set of parentheses which immediately follows it, and the usual name scoping rules apply wherever the iteration construct nests such variables. The use of the iteration construct can be found in production rules 3, 6, 21, 25, 27, 30, 33, 38, 40, and 43, where the construct is always of the form:

SYMBOL SUBSCRIPT`(´SUBSCRIPT DELIMITER α`)´

and α stands for a series of one or more substrings all separated by the same delimiter as that preceding the α. (If the substrings are themselves iterator constructs then they must be derived by the same production.) Iteration implies merely a shorthand notation for writing a list of items separated by some specified delimiter, for example, the `x´ in production (30). If we call the integers associated with the subscripts in the iterator construct s_1 and s_2, then the number n of items in the

(implied) expanded list which iteration indicates is s_2-s_1+1 for $s_2>s_1$ and zero otherwise. If the iterator symbol does not appear in α then we indicate a list of n identical items. If the iterator does appear in α then we indicate a list of n items formed by textual substitution of digits and/or subdigits (see productions (10) and (11)) for the iterator symbol(s) in each item. More specifically, the strings representing the integer values

$$s_1, s_1+1, s_1+2, \ldots, s_2-1, s_2$$

are substituted for all occurrences of substrings matching the iterator symbol in the respective n items in the expanded list. The matching substrings may occur within symbols or subscripts. Each symbol or substring is scanned from left to right for non-overlapping occurrences of the substring matching the iterator symbol. Then all substitutions of integers for symbols are made simultaneously to yield the new string. For example the list a_1, a_2, a_3, a_4 can be represented as simply $i_1(_4, a_i)$, where the iterator symbol i occurs as a subscript. Other examples of textual substitution of integers for iterator symbols can be found in section 3.4.3.3.2.2 in connection with the semantic base language representation for the iterator construct.

(5) A name is denoted by a symbol which may be accompanied by one or more subscripts separated by commas (themselves subscripts, as opposed to ordinary commas). Two identical symbols with different subscripts or different numbers of subscripts are considered as distinct names. Wherever the iterator construct occurs, of course, we are referring to the textually substituted subscripts implied by the iterator symbol rather than the symbol itself. That is, we consider the iterated list to be fully expanded for the purposes of name identification.

(7) Symbols may not be any of the following reserved terminal strings: `TRUE´, `FALSE´, `BOOLEAN´, `INTEGER´, `CHARACTER´, `STRING´, `WITH´, `LET´, `BE´, `EB´, `WHERE´, `XC´, `XA´, or `XS´. The nonterminal ALPHABET is not defined here since we have not chosen a particular character set. However, it may be rewritten as any one of a set of terminals which includes at least the letters of the alphabet and which shares no characters in common with sets of terminals for DIGIT, SUB_DIGIT, SUB_ALPHABET, or DELIMITER. In this report we do not distinguish between capital and lower case letters.

(8-9) A subscript may be either an integer or a symbol. A symbol used as a subscript must correspond to an iterator symbol or to a named constant. Again, as for ALPHABET,

the set of terminals associated with SUB_ALPHABET is not specified. However, this set does not have any characters in common with ALPHABET, SUB_DIGIT, or DELIMITER. In this report all characters in the set for SUB_ALPHABET are written as subscripted versions of ordinary letters of the alphabet.

(12) Attributes are uninterpreted comments which may be appended to definitions.

(13-14) The left quote and right quote always occur in pairs and are neither delimiters (see production (15)) nor non-delimiters (see production (16)). Hence the non-terminal ALPHABET cannot be rewritten as either quote mark.

(20) A specification consists of a set of one or more state successor functions corresponding to a set of possibly interacting asynchronous processes. Processes interact, of course, via exchange functions. No interactions are possible between processes defined in different specifications since exchange classes have scope only within the specification in which they are defined.

(22-30) If no set expression occurs in production (22) then the set named in that production is primitive, that is, not specified in the source language. Otherwise, the

rules for forming sets permit only explicit definitions of sets in terms of literals and named constants (productions (24-26)), in terms of unions of sets defined elsewhere (productions (27-29)), or in terms of Cartesian products of sets defined elsewhere (productions (28-30)). (Named constants (production (26)) differ from literals in that they have no explicitly assigned value; they are declared as distinct entities by their appearance in an item list.) There is no operator precedence associated with the symbols `∪' and `×' and the syntactic rules prohibit the mixing of the symbols in a set expression without the use of parentheses. For example, A∪B∪C and A×B×C are valid set expressions but A∪B×C and A×B∪C are not. The following, however, are legal set expressions: (A∪B)×C, A×(B∪C), A∪(B∪C), and (A×B)×C. The union operator is, of course, both commutative and associative so that the following set expressions are equivalent: A∪B∪C, (A∪B)∪C, and A∪(B∪C). On the other hand, the Cartesian product operator is neither commutative nor associative, so that no two of the following set expressions are equivalent: A×B×C, (A×B)×C and A×(B×C). Superfluous pairs of parentheses are ignored. Thus the set expressions (A×(B)), (A×B), and A×B are equivalent.

(31-38) A primitive function is specified merely by its name, the two set expressions for domain and range, respectively, and possibly some attributes. That is, no mapping between the sets is given. However, for non-primitive functions we have in addition a parameter list, which must be of the same data type as the elements in the domain of the function, and a functional expression, which must evaluate to a form which is of the same data type as some of the elements in the range of the function. (By data type we mean the tuple structure of set elements defined by Cartesian products of other set elements.) A single name always suffices as the sole parameter in the parameter list. However, if the elements in the domain of the function are defined by Cartesian products of sets then the parameter list may be written in any (syntactically correct) parenthesized fashion such that each parameter and parenthesized list matches a set or set expression in the domain definition. In a similar fashion each function invocation (see production (42)) and parenthesized function tuple (see production (38)) in the functional expression (see production (35)) defining the function must correspond to a set or set expression in the range of the function. A few examples will help to clarify these statements. If we consider the function f such that

f: $(A\times(B\times C))\times D \rightarrow E\times(G\times F)$,

then the following represent a few possible mappings for defining f:

$f(x) \equiv g_1(x)$

$f(x) \equiv (g_2(x), g_3(x))$

$f(u,v) \equiv g_4(v,u)$

$f((w,(y,z)),v) \equiv (h_1(z),(h_2(w),h_3(y,v)))$

where it is assumed for consistency that other definitions in the specification allow us to infer that dom f $\subseteq$ dom g_1, $g_1(x) \in E\times(G\times F)$, dom f $\subseteq$ dom g_2, $g_2(x) \in E$, dom f $\subseteq$ dom g_3, $g_3 \in G\times F$, $D\times(A\times(B\times C)) \subseteq$ dom g_4, $g_4(v,u) \in E\times(G\times F)$, $C \subseteq$ dom h_1, $h_1(z) \in E$, $A \subseteq$ dom h_2, $h_2(w) \in G$, $B\times D \subseteq$ dom h_3, and $h_3(y,v) \in F$. By definition the parameters u, v, w, x, y, and z refer to elements in $A\times(B\times C)$, D, A, $(A\times(B\times C))\times D$, B, and C, respectively. Parameter names (but not commas) may be omitted from the parameter list (see production (34)) if they do not appear in the functional expression in the function definition. For example, a constant function needs no parameters. Finally, we should note that parameters have scope only within the function definitions in which they occur. Also the repetition of a parameter name in a parameter list is inconsistent and so is not permitted.

(39-41) A selector function allows the conditional evaluation of functional expressions. The first member in each pair of functional expressions (production (41)) must evaluate to a Boolean result. Evaluation of the selector function itself goes as follows. The first member of each pair of functional expressions is evaluated in turn from the leftmost pair to the rightmost until the value TRUE is encountered for some expression. Then the second expression in that pair is evaluated and constitutes the value of the selector function. If no TRUE value is encountered in any pair of expressions then the final expression in production (39) is evaluated instead and the result becomes the value of the selector function. No functional expressions in the selector function are evaluated except for those mentioned above.

(42-43) A function invocation is either a literal or a list of one or more function names (indicating composition of functions) with an expression list. In the latter case the evaluation of the function(s) and expression list follows the ordinary rules of algebra. The expression list must, of course, evaluate to some data type which conforms to the elements in the domain of the last function named in the name list. Similarly for each contiguous pair of function names in the name list the range of the second function must be contained in the

domain of the first function. Note that the expression list is empty only when the last function in the name list has no parameters in its definition, that is, is a constant function. However, parentheses are required following the name list even for a null expression list so that function names and parameter names are always syntactically distinguishable. Elements in the expression list are evaluated in indeterminate order, possibly in parallel, precisely once, regardless of whether they have a corresponding parameter name in the definition of the function. Naturally, when expression lists are nested the ordinary precedence rules of algebra apply. Namely, all of the arguments of a function must be evaluated before the function itself can be evaluated.

A final addendum pertaining to the specification source language is that blanks may be inserted anywhere except within symbols, strings, or integers without altering the syntax. We require that reserved symbols (see note (7) above) be preceded by one or more blanks and that unsubscripted reserved symbols (i.e., those other than XC, XA and XS) be followed by one or more blanks.

3.4.3.3 Semantic Base Language

Our methodology consists of at least the following:

-- a set of procedures for producing specifications

-- automated tools to support the procedures

-- rules as to the applicability of each procedure and tool

Using the methodology, one starts with a "cloud" of vague, informal, incomplete, possibly inconsistent or infeasible "ideas" of the desired behavior and generates an axiomatic model. In the next stage, the axiom set and cloud are used to generate the first effective, through abstract specification. One then repeatedly applies the methodology to elaborate the specification to arbitrary detail. From the first effective specification onward, the design may be factored by any of a number of techniques and given to several designers for separate development. Each group may have its own set of notational and managerial preferences, but each must produce a specification which is, by nature of the earlier factorization, a portion of the overall system specification.

In this section we describe a common design data base representation language (a semantic base language) that will provide the object structures to be managed in an implementation of our methodology. The Semantic Base Language (hereafter called SBL) will serve as the common means for representing specifications for the proposed Computer Assisted Formal

Engineering Laboratory. It is envisioned that several user or application specific languages will evolve in which to express specifications and these will all be transted into the SBL. In this way, diverse groups of designers may use their own favorite notation to produce a specification which is part of the overall system specification and may be tested and analyzed with the same common tools. A set of "de-translators" will also exist to map the base language into a form familiar to the designer. In this way, diverse designers can see a specification in a manner they are familiar with. It is also envisioned that interpreting procedures will use the base language to simulate directly the system and display the dynamic behavior. Figure 3.4-1 depicts these attributes.

We will describe SBL by a formal grammar which states rules for forming sentences in the language. These rules are called productions of the formal grammar and only represent syntactic rules (i.e., the form of sentences in the language). One may say something that is syntactically allowed in the base language but is semantically disallowed (i.e., it can be said in the base language but has no valid meaning in the context of a specification).

It is our goal to explain informally the semantics of each production in terms of common algebraic concepts.

In order to specify the production we have adopted (with slight modification) a set of notational rules proposed by Wirth and given in Appendix C.

3.4.3.3.1 SBL Principles

A brief discussion of some of the design goals for SBL may help in understanding the SBL structure described in the next section.

3.4.3.3.1.1 Content

The SBL language consists of definitions (primitive and nonprimitive) of sets, functions, specifications, and blocks. Each such definition must explicitly code the corresponding source language definitions so that a de-translator from SBL to source language can be constructed. This property ensures that the required specification properties are testable and preserved in SBL.

3.4.3.3.1.2 Block Structured

The definitions in a specification should be collected into blocks in order to control their scope and allow distribution of blocks over a network of interacting design laboratories. These block structures should be nested so that blocks may be themselves split into subblocks.

The definitions in the nested blocks should have extent (in the normal Algol way) so that designers may freely choose their definitions without conflicts in symbols. As a consequence, we will use the Algol block and declaration structures as the basis for SBL. We include here a brief discussion of block structure concepts.

A block structure creates a tree-structured environment for elements in the specification such that each element has a unique "range" over which it applies. Any element defined in a given block is known to innermore (leafward) blocks and unknown to outermore (rootward) blocks. Algol scope and extent rules specify which elements are applicable as you move from environment to environment. Refer to Figure 3.4-2 for the following discussion of block structured, Algol-like scope and extent.

The outer most block (root) is block A which has two innermore blocks (branches) B and C. Block C contains the innermore block D.

In block A function X is defined, its scope is all of block A including blocks B, C and D (since they are in A). Similarly functions Y and Z defined in blocks B and C respectively have B and C as their scope. Note, however that Z is also defined in block D. This causes the meaning of Z to reflect the definition of Z at the beginning of block D whenever a reference to Z appears inside block D. This means that any reference to Z outside of block D must have been defined elsewhere in the current environment, as specified by the block nesting.

The environments in the example are:

A: A only

B: B (innermost) - A (outermost)

C: C (innermore) - A (outermost)

D: D (innermost) - C (outermore) - A(outermost)

Anything not defined in present block must be defined in an outermore block. Note also that anything defined in an inner block is unknown to an outer block.

3.4.3.3.1.3 Machine Independence

The SBL should be machine independent so that the design laboratory and the methodology may be readily transported to design subcontractors and used on many projects. We will define all of our methodology in terms of SBL constructs. A new implementation will require only an implementation of an interpreter for the definitions. It is essential that designers be insulated from individual implementation idiosyncracies since designs produced on many different implementations will be integrated into the final specifications.

We will require the SBL to be recursive in the sense that a specification can be treated as a data object in another specification. We can thus design our laboratory and define its structure in the same way that we design application systems. Evolutionary operations may then be defined in the same formalism.

3.4.3.3.1.4 Tuples

We will use the abstract algebraic structure called tuples to represent sentences in SBL. In effect, tuples represent an abstract data type and our design laboratory will be defined in terms of that data type. Each tuple will be typed so that it is possible locally to identify the context and proper interpretation of definitions. Thus SBL will be a strongly typed language. General strong typing implies that one knows the type of an element simply by inspection. In the case of the base language, all tuples will contain as their first element a unique tuple type tag, hence very strong typing is used.

In addition, the base language will be represented by a context-free grammar. Thus for each production there will be only one nonterminal on the left side, implying that there are no places where a nonterminal's meaning depends on the context in which it is used.

3.4.3.3.2 Semantic Base Language Description

We shall proceed in the description of the semantic base language in a bottom-up fashion. We give a precautionary note that some nonterminals do not derive terminal strings. This is due to one of two cases: (1) the productions are dependent on the particular character set chosen, or (2) the exact terminal string(s) derived are not

important to the present discussion. Some examples of the second case are the different nonterminals for the different tuple types. The exact coding for a particular type indicator (for example BOOLEAN_TYPE) is not important to the present discussion.

3.4.3.2.1 LITERALS

```
LITERAL = `(´BOOLEAN_TYPE`,´BOOLEAN`)´|
          `(´INTEGER_TYPE`,´INTEGER`)´|
          `(´STRING_TYPE`,´STRING`)´.
BOOLEAN = `TRUE´|`FALSE´.
INTEGER = DIGIT{DIGIT}.
  DIGIT = `0´|`1´|`2´|`3´|`4´|`5´|`6´|
          `7´|`8´|`9´.
 STRING = ``´{STRING_ELMT}``´.
STRING_ELMT = NON_QUOTE|STRING.
```

A literal is a boolean value (TRUE or FALSE), an integer value (sequence of digits), or a string. A string is a sequence of characters and (sub) strings enclosed within quotes. The nonterminal NON_QUOTE has not been expanded; it should derive every character in the character set except left and right quote marks. Note that left and right quote marks are not permitted in a string except to indicate substrings.

3.4.3.3.2.2 Names

```
NAME = SYMBOL|`(´NMT`,´SYMBOL[`,´SUFFIX_LIST]`)´.
SYMBOL = NON_DELIMITER{NON_DELIMITER}.
SUFFIX_LIST = (SUFFIX|`(´ITT`,´ITERATOR_CONTROL`,´
              SUFFIX_LIST`)´){`,´SUFFIX_LIST}.
SUFFIX = SYMBOL|INTEGER.
ITERATOR_CONTROL = SYMBOL`,´SUFFIX`,´SUFFIX.
```

A symbol is a sequence of characters which does not include one of the delimiters `(´,`)´, `,´, ```, or `´´. We place a further semantic restrictions that a symbol is not an integer nor is it one of the reserved symbols `TRUE´, `FALSE´, `BOOLEAN´, `INTEGER´, `CHARACTER´, or `STRING´. Now a name is either a symbol as described above or a name tuple. A name tuple is a subscripted symbol. The suffix list provides the list of subscripts. Before we describe the suffix list we will explain the iterator construct (which, as we shall see, provides a macro-like facility).

Often we are required to list a large number of items, for example a large number of suffixes. An explicit listing would be cumbersome, so we add a facility to list the items in a rather concise fashion. (An analogy can be drawn to Fortran which provides an implied DO-loop for listing array elements in a rather concise fashion.) Here we will provide the iterator construct.

The iterator consists of an iteration control followed by a list which is to be iterated over. The iteration control consists of an index (the symbol) which is like the loop index in Fortran. The next two items are suffices indicating the starting and ending values of the index. These suffixes for starting and ending values must be either an integer constant or a symbol which is an index for a containing iterator. (Iterators may be nested, as may implied DO-loops in Fortran.) With this restriction on suffixes we can look merely at the iterator to tell the number of iterations; no "run-time" values are needed.

The iterator generates a list of items (suffixes in this special case). For each value of the index in the specified range, a copy of the suffix list is added to the iterator-generated list, with each textual occurrence of the index replaced by the text for the integer value. (If the ending value is less than the starting value the list generated is empty.) The replacement is done in a left to right fashion. Some examples may help clarify the situation.

```
(ITT, I, 1, 10, I)
    is equivalent to 1,2,3,4,5,6,7,8,9,10

(ITT, I, 1, 5, `SI´, SI)
    is equivalent to `S1´,S1,`S2´,S2,`S3´,S3,
    `S4´,S4,`S5´,S5
```

(ITT, M, 1, 2, (ITT, N, 1, M, NM, MN))

is equivalent to (ITT,N,1,1,N1,1N), (ITT,N,1,2,N2,2N)
is equivalent to 11, 11, 12, 21

We can now continue with a description of the suffix list. A suffix list is a list of suffixes and iterators (over suffix lsits). A suffix is a symbol, an integer constant, or a named constant.

3.4.3.3.2.3 Sets

```
SET = `('SET_TYPE`,'NAME[`,'SET_EXP][`,'ATTRIBUTES]`)'.
```

Here we have a set definition. Each set has a unique name. The set expression, if present, gives a definition of the set (as explained below). If absent, the set is considered to be a primitive set.

```
ATTRIBUTES = `('A_TYPE{`,'ATTRIBUTE}`)'.
 ATTRIBUTE = `('USER_ATYPE`,'STRING`)'.
USER_ATYPE =  NON_DELIMITER{NON_DELIMITER}.
```

Attributes are informal comments that do not affect formal properties of specifications. The tuple of attributes is given a type via our nonterminal ATYPE. (Remember we have not and will not expand it here.) For the tuples within this one the user is free to define his types. However, we must make a semantic restriction that the type not be one of the

types we use in our semantic base language

```
SET_EXP = SET_CROSS|SET_UNION|SET_INVOC|SET_FORMER.
```

A set expression is used to produce a set result. Set cross produces the cross product of the two or more sets. Set union produces the union of two or more sets. Set invocation allows us to use previously defined sets. Set former allows us to construct a set by listing its elements.

```
SET_CROSS = `('CROSS_TYPE`,'SET_LIST`)'.
SET_UNION = `('UNION_TYPE`,'SET_LIST`)'.
SET_LIST  = (SET_EXP|`('ITT`,'ITERATION_CONTROL`,'
            SET_LIST`)'){`,'SET_LIST}.
```

Here we define cross products and unions of a list of sets. A set list is a list of set expressions and iterators over set lists. We place a semantic restriction that the set list contain at least two sets.

```
SET_INVOC = `('SINVT`,'NAME`)'.
```

Here we invoke an intrinsic set (for example integer) or a set previously defined by the user of our formalism. The set invocation will be used in a set expression.

```
SET_FORMER   = `('FORM_TYPE`,' ELEMENT_LIST`)'.
ELEMENT_LIST = (SYMBOL|LITERAL|`('ITT`,' ITERATION_CONTROL`,'
               ELEMENT_LIST`)') {`,'ELEMENT_LIST}.
```

Here we explicitly construct a set by listing its members. Its members may be literals or symbols, which are interpreted as named constants. Named constants follow normal name scoping rules. The appearance of a named constant in a set former is an implicit declaration of that named constant.

3.4.3.3.2.4 Functions

```
FUNCTION = `(´FUNCT`,´ NAME [`,´ PAR_LIST] `,´
           SET_EXP`,´ SET_EXP [`,´ FUNC_EXP]
           [`,´ ATTRIBUTES]`)´.
```

Here we have a function definition. Name is the function name, followed by a (possibly empty) list of parameters, followed by two set expressions giving the domain and range, followed by a definition of the function's computation, followed by some informal comments. If the function expression is not present, the function is considered to be a primitive function.

```
PAR_LIST = ([NAME]|`(´ITT`,´ ITERATION_CONTROL`,´
           PAR_LIST`)´|`(´PELMT`,´PAR_LIST`)´)
           {`,´PAR_LIST}.
```

Here we list the formal parameters for the association function. Name is optional because we may not want to name part of a domain. An example of this is the projection function

f: A × B → A where f(a,b) = a. The second argument need not be named since it is not used, as can be seen in the following specification of f.

```
(FUNCT, f, (PELMT, a, ),(CROSS_TYPE, (SINVT, A),(SINVT,B)),
     (SINVT, A) , (PARINVT, a))
```

Also note that the nesting of parenthesis is important. That is, if f: A × (B × C) × D → E, then a description of arguments should be (PELMT, a, (PELMT, b,c),d) and not (PELMT,a,b,c,d), which would be proper if f: A × B × C × D → E.

```
FUNCT_EXP = PAR_INVOC|SELECTOR|FTUPLE|FUNCT_INVOC.
```

A function expression is simply a parameter (see example of projection function above), a selector function (a functional form of a case statement with an otherwise part), a tuple of function expressions, or an invocation of another function. Before we continue and elaborate the permissible forms of a function expression, we will define the semantics of function evaluation. In order to evaluate a function f(E1,E2,...En), all its arguments E1,E2,...,En are evaluated first to come up with associated values V1,V2,...,Vn. Then V1,V2,...,Vn are used in the defining computation. So our evaluation of functions resembles call by value. Notice that the reason that we are forced to discuss this issue is that our functions may have side effects (because of asynchronous

interactions via exchange functions), so we must specify when and how many times an argument is evaluated. Without side effects these considerations are not important.

```
PAR_INVOC = `('PARINVT`,' NAME`)'.
```

Here we invoke a parameter. The result of a parameter invocation is the value of the associated parameter (as described in the previous paragraph).

```
SELECTOR = `('SELT [`,' PAIR_LIST]`,'FUNCT_EXP`)'.
PAIR_LIST = (PAIR|`('ITT`,' ITERATION_CONTROL`,' PAIR_LIST`)')
            {`,' PAIR_LIST}.
PAIR = `('PAIRT`,' FUNCT_EXP`,' FUNCT_EXP`)'.
```

A selector function consists of a list of pairs of function expressions followed by a single function expression. We can use the iteration construct to list some of the pairs. A pair consists of two functions; the first is a boolean function, the second a function that produces a result in the proper range for the selector function.

The selector function is a functional form of the case statement with an otherwise part. Let [B1:E1,...,Bn:En,En+1] be a selector function. For $1 \leq i \leq n$, Bi:Ei is a pair where Bi is a boolean function and Ei is a function whose range is the same as Ej for $1 \leq j \leq n+1$. The evaluation of the selector function is done as follows. The B's are evaluated left to

right. If no Bi evaluates to TRUE then En+1 is the result of evaluation. Otherwise let Bi be the first B that evaluates to true. The result of evaluation is then Ei. Note that Bj is not evaluated where $i < j \leq n$.

```
FTUPLE = `('FTUP`,' FUNC_EXP_LIST`)'.
FUNC_EXP_LIST = (FUNC_EXP|`('ITT`,' ITERATION_CONTROL`,'
                FUNC_EXP_LIST`)')[`,' FUNC_EXP_LIST].
```

Here we define a function tuple, which is just a list of function expressions. Again we can use an iterator to list some of the function expressions. The value of the function tuple (E1,...,En) is (V1,...,Vn) where Vi is the result of evaluating function expression Ei, $1 \leq i \leq n$.

```
FUNCT_INVOC = LITERAL|`('FINVT`,' NAME_LIST [`,'FTUPLE]`)`.
NAME_LIST = (NAME|`('ITT`,' ITERATION_CONTROL`,'
            NAME_LIST`)`)[`,'NAME_LIST].
```

Here we have a function invocation. A function invocation can be literal, that is an expression whose result is always the same. In the second choice for the function invocation production, the list of names are functions which are composed to operate on the arguments (the function tuple). The function tuple is optional as some functions can have no arguments (for example, a function that reads a clock). An example of a function invocation would be f∘g∘h(x) (or equivalently

f(g(h(x)))), which would be encoded as (FINVT,f,g,h,
(FTUPT, (PARINVT, x))).

3.4.3.3.2.5 Specifications

```
SPECIFICATION = `(´SPECT`,´NAME [`,´F_LIST]
                    [`,´ATTRIBUTES]`)´.
FLIST = (FUNCTION|`(´ITT`,´ITERATION CONTROL`,´
                    F_LIST`)´) {`,´F_LIST}.
```

A specification is a named listing of asynchronously interacting processes. The list of functions are state successor functions for processes (and thus have the domain and range identical). We place a semantic restriction that there are no inter-specification exchange classes.

Given initial values for the domains of the processes in the specification, we could then simulate the behavior of the processes by applying the state successor functions. Note again that the processes are asynchronous.

3.4.3.3.2.6 Blocks

```
BLOCK = `(´BLOCKT`,´NAME [`,´DEF_LIST]
                [`,´ATTRIBUTES]`)´.
DEF_LIST = (DEFINITION|`(´ITT`,´ITERATION_CONTROL`,´
                DEF_LIST`)´) {`,´DEFINITION}.
```

A block is a named list of definitions (to be described below). Thus we can provide Algol-like name scoping rules.

3.4.3.3.2.7 Definitions

DEFINITION = SET|FUNCTION|SPECIFICATION|BLOCK.

This is the start symbol of our data base representation language. With this representation language we can encode set, function, and specification definitions with an Algol-like block structure. The representation language was chosen to be an encoding that is easily manipulated by machine and not as a language to be written and read by humans. The actual strings in the representation language are typed tuples as described by the above productions.

3.4.4 Computer Assisted Formal Engineering Laboratory (CAFEL)

The specification semantic data base becomes the object of our DDP methodology. The implied computer system must support a large, distributed, multi-user, data base that can easily evolve with our design theory.

The data base will be large because of the scale of the systems being designed. The data base must be distributed since many contractors may be involved in the associated development processes. Many designers may be operating on the data base in parallel and distributed control must be provided to maintain integrity and consistency of the specifications.

We may use our specification methodology to develop this CAFEL. We have only begun such a development and most of the design work remains to be done. Even so, the exchange graph characterization of the proposed CAFEL design at a high level of abstraction provides an interesting example of a distributed, hierarchical, real time data base system.

3.4.4.1 Buffered Interactions

We will require the processes of CAFEL to operate as free running with only XS type of interactions. Thus they may be distributed freely without design constraints. In effect, each process is autonomous in the sense that any local operation may be done locally no matter what the other processes may do. Synchronization between such processes is voluntary. We may

accomplish this autonomy by the use of a standard buffer process.

The buffer process is parametric and for each value of the parameter, a new buffer process will be generated in the specifications. The buffer process is defined in Figure 3.4-3.

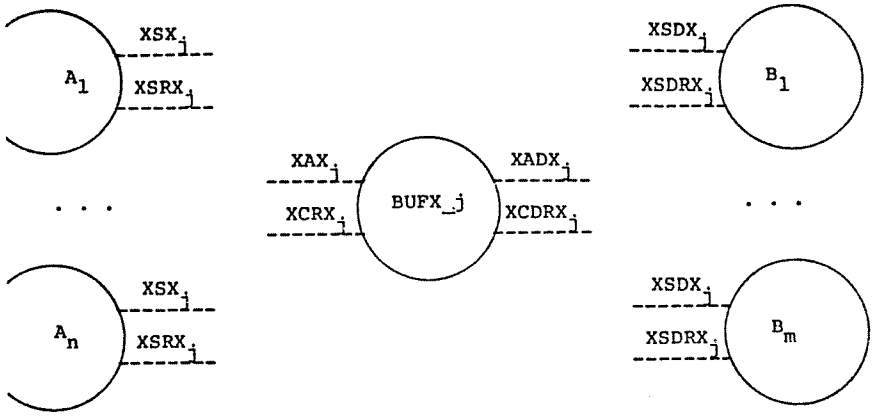
3.4.4.2 User Processes

The user processes in Figure 3.4-5 are mere shells that specify only that they generate commands for the data base manager processes. Their elaboration may be user dependent. The user process will be mapped by the local implementation onto a "command channel." Several users may cooperatively share a command channel. The control of access will be via control of command channels. There will be a unique "right to receive" for each command channel that will exist in some data base manager (DBM) process. Users of that command channel will access directly only the DBM that has the corresponding right.

Each command channel may have at most one outstanding command and the channel j buffer, BUFC_j, will not accept a new command until the results of the previous command have been returned to the originating user.

3.4.4.3 Data Base Managers

The data base managers contain the semantic data base, which is a tree structure of nested blocks. The data base



$$BUFX\_j \equiv XCRX_j(XADRX_j(XADX_j(XAX_j(FALSE))))$$

Figure 3.4-3. Generic Buffer Process Model

The processes "A_i" will compete for the buffer by executing XSX_j until the buffer accepts one of them. The buffer will then provide the message to the first "B_j" process that executes $XSDX_j$. A response by "B_j" is then returned to the buffer and then transferred to the originating "A" process. The buffer process is simply (BUFX_j,) with a null state space.

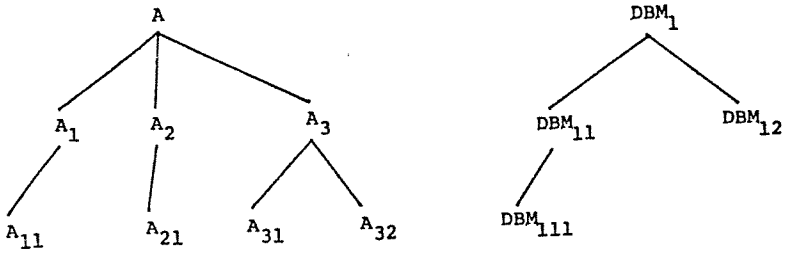


Figure 3.4-4. Semantic Data Base Partitioning.

A possible distribution of the blocks over the DBM is given by the following:

DBM_1 has A and A_2, DBM_{11} has A_1 and A_{21}, DBM_{12} has A_3, A_{31}, and A_{32}, DBM_{111} has A_{11}.

managers will also be tree structured and the semantic tree blocks will be mapped onto the DBM tree while preserving the nesting relationship. An example of this mapping is given in Figure 3.4-4.

The definitions in a given DBM are in terms of other definitions in that DBM or one of its ancestor DBM. Thus all definitional environments extend rootward in both the DBM and block trees.

The command channel "rights" are originally vested in the root DBM and are allocated to child DBM by the local policy of each parent DBM. A DBM that has suballocated such channel rights will not make any modifications in its associated blocks. This constraint will prevent update conflicts between DBM, and ensure an invariant environment down to the level of the DBM that is making local modifications.

Each DBM will have a $BUFM_k$ buffer process where k is the index for the parent DBM. This buffer is used for requesting information from the more global data base blocks contained in ancestor DBM. All of the siblings will compete for the use of this parent buffer. At most one parent request will be serviced at a time.

3.4.4.4 Service Processes

Each DBM may require a set of service processes to carry out its commands. The requests for service will use BUFV_sm

where s is the service and m is the DBM. At most one outstanding request for each service by each DBM is possible.

In turn, a service process may need information from its associated DBM (or its ancestors). The BUFS_m will be used and at most one such request by the set of service processes associated with the m DBM will be permitted.

3.4.4.5 Exchange Graph

The exchange graph of Figure 3.4-5 formally defines the relationships between the processes. All of the possible interactions are explicitly shown. The process specifications themselves are not shown and must be developed. The exchange graph has allowed the early recognition and resolution of the distributed, multi-user, and hierarchical properties of the data base system. Subsequent design decisions can be made on a local basis and will not violate those described in the exchange graph. This design also provides minimal constraints on the local design decisions while allowing significant parallelism and distribution in the design data base management. This is the beginning of the CAFEL design and a framework for subsequent design.

3.4.5 Specification Analysis

Our formal specification language has the properties we required. We give a brief description of why each of the properties hold.

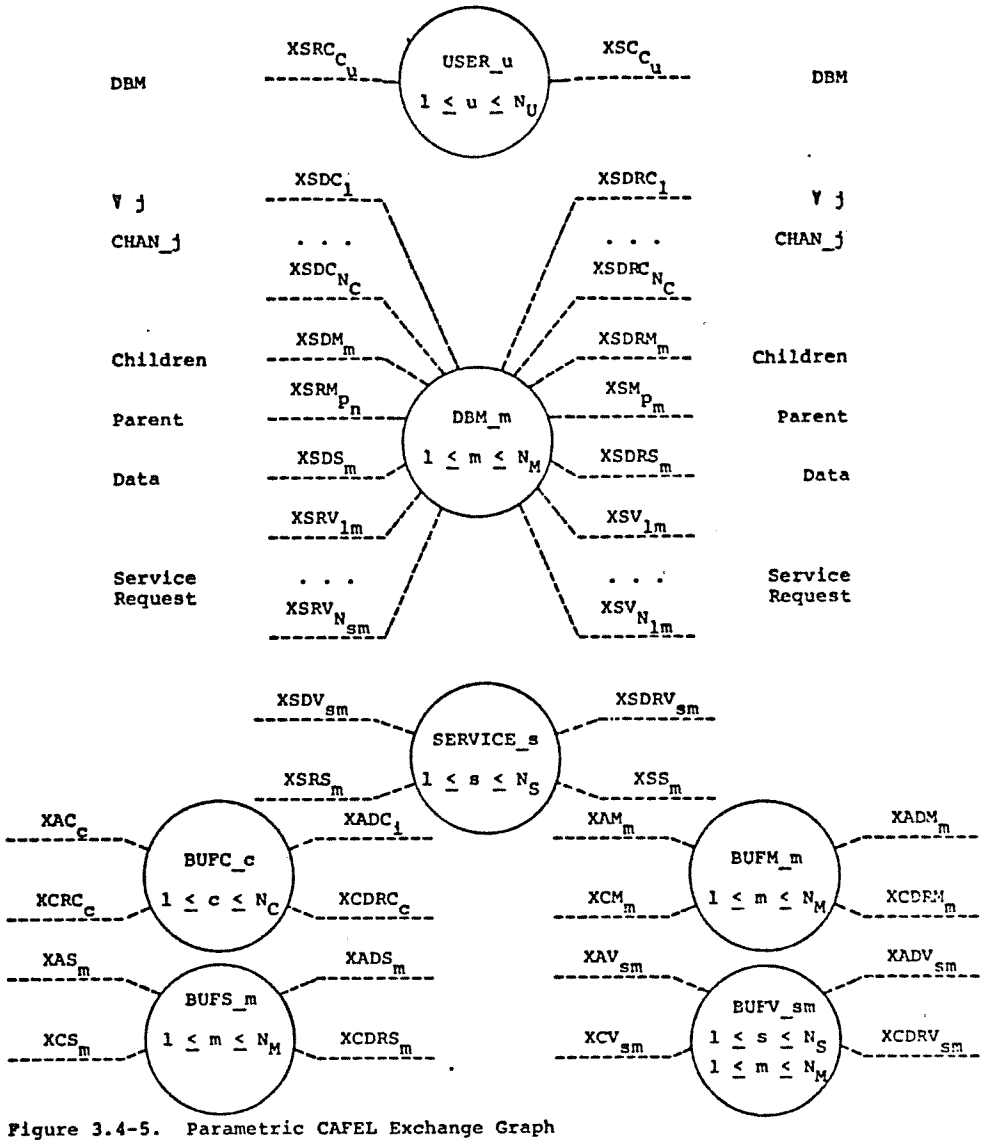


Figure 3.4-5. Parametric CAFEL Exchange Graph

The parameters are N_M, N_U, N_S, N_C and each parameterized process represents a process for each value of its index. Exchange functions are connected by dashed lines to all others in the same class. The connecting lines are suppressed for clarity.

BUFC_c is used for commands from users to DBM.
BUFM_m is used for commands from children to parent DBM.
BUFS_m is used for commands from services to the associated DBM.
BUFV_sm is used for commands from DBM to associated services.
P_m is the index of the parent DBM of DBM_m.
C_u is the CHAN_j assigned to user u.

3.4.5.1 Formality

A specification in our specification language is a sentence in a decideable context-free language. Therefore it is a formal specification.

The test for being in the context-free language is simply a parsing algorithm that can readily be automated. The existence of syntactic errors can be detected, analyzed, and displayed to the designer, just as for conventional programming languages.

3.4.5.2 Consistency

A specification will be consistent if it is interpretable as a system. We must require the definition of each symbol, the matching of function domains with the corresponding argument, and the matching of the state space with the process successor function.

Provided these conditions are met, the process specification is readily interpreted as a system generator, and thus consistency holds.

The checks for these conditions may easily be made by algorithms built into the parser for the context-free language. All inconsistent conditions will be detected, analyzed, and displayed to the designer.

3.4.5.3 Effectiveness

The specifications will be effective if there is a universal interpreter for the state successor function that is an algorithm. The only condition preventing that interpreters existence is that of "logical blockage" of an exchange function. Such a blockage would prevent the interpreter from finishing the state successor function evaluations. Sufficient conditions for testing for exchange blockage, are given in Appendix A. If these conditions are met, the specification will be testable for the algorithmic property.

The analysis for this property will detect failures of synchronization at an early development stage or will certify that interprocess interactions are consistently synchronized.

An important by-product of this property, is that a specification is its own simulation model. It can be automatically translated to executable code that will generate any desired computation of the specified system. The designer may thus carry out simulation experiments, confident that his experiments match his designs.

The simulation experiments may include the "real world" processes that drive the designed DDP system. The same design theory and specification language can be used to specify the desired "real world" processes and thus form a closure with the control processes.

Clearly, the development of the desired "real world" behaviors, the desired control system behaviors, and simulation models must all be carried out in parallel. With our specification language, these objectives can all be achieved in the same specification.

3.4.5.4 Homogeneity

The operator, E, can define a primitive function or set as a consistent expression in a specification. The new specification will be for a system that is contained in the more general initial specification. The homogeneity of our specification language follows immediately.

The automation of the operator E will provide a way for a designer to introduce elaborations of primitives to a lower level of abstraction (i.e., in terms of more basic primitives). The consistency of this design change is ensured. The previous design decision will still be valid since the effect of the elaboration will be only to restrict computations in the initial set to a subset that is the new system. No new behavior will be introduced. Undesired old behavior can be eliminated. This operator alone will allow substantial development progress with completely automated validation.

3.4.5.5 Modularity

Our specification language is modular with respect to asynchronous processes. The synthesizing operator is the

asynchronous composition and the elaboration operator is just as above.

Our specifications are hierarchically modular with respect to the definitions of functions. The elaboration can be the same as above and the synthesizing operators are the allowed forms of functional expressions.

As a result of this property we can partition specifications into modules that can be locally designed. This will be important for our subsequent methodology.

3.4.5.6 Informal Extensibility

The provision of attributes for each definition in a specification is sufficient to produce the extensible property.

Our formal analysis tools will not analyze these attributes but CAFEL will preserve them and make them available for any operations specified by designers.

3.4.5.7 Distributed

Our specifications include asynchronously interacting processes. Any distributed system can thus be specified with explicit requirements for synchronization and interprocess control at any desired level of abstraction.

The simulation of such specifications will allow study of the intrinsic distributed data processing system properties and behaviors.

3.4.6 Simulation

An important result of specifying systems using our formalism is that the evaluation of a specification applied to a set of initial states for the component subsystems is well defined. A specification may be interpreted (or compiled to implementing procedures) as a generator of computations. Thus a specification at a suitably detailed level could be used to produce automatically its own implementation via a suitable compiler. In effect the distinction between system and program design is one of level of detail, and of binding to either hardware, software or firmware by the implementing compiler.

The simulation (generation of computations) of a specification can proceed automatically to the level of detail provided by the primitive functions and sets used in that specification. There are several types of simulation possible, based on the interpretation of primitives in the specification. The values produced by reference to primitive functions may be obtained at each of several levels of approximation.

The first level is to interpret a primitive function by making a stochastic selection of a value (with given probability distribution) from its range. Thus, at this level, all primitives can be simulated by the same parameterized procedure.

The next level is to provide some approximating procedure for the primitive function that can be used to generate function values.

A third level is to provide an exact (but possibly not optimized) procedure to generate function values. If the specifications have been elaborated to very low level primitives, the specification of the corresponding procedures will permit the implementation of the system by a "compiler."

The conventional discrete event simulation technique requires that all "events" must be mapped into instants of simulated time and each "event procedure" must be evaluated in a linear sequence of such procedures. Thus a distributed system is simulated by a central (sequential) system in which all asynchronous conflicts have been resolved as part of the definition of the simulation experiment. This may easily lead to the omission of critical distributed behavior problems from the model. Certainly, the quantum of time resolution may be decreased and, at great inefficiency, a "correct" simulation may be produced. However, the inefficiency is frequently prohibitive. A distributed system requires also a distributed simulation. The conventional centralized simulation languages make distributed implementation of the simulating system very difficult. Our formal specification language is also a distributed simulation language.

We must design our simulator so as to permit distributed implementation. Indeed, in the limit, the simulator system will simply generate an implementation of the specified system. We will design our simulator using our formal methodology.

Thus a CAFEL may be formally specified and a prototype simulator may be used to generate a new implementation.

3.4.7 Conclusions

We have designed a prototype formal specification language that has at least the formally required properties developed in the previous sections. This language is a slight extension to ordinary algebraic notation and can specify asynchronously interacting processes with great generality.

We have started the design of a Computer Assisted Formal Engineering Laboratory (CAFEL) that will provide the semantic base for the development and analysis of specifications.

Algorithms for the testing of the specifications can be constructed in terms of the semantic data base structures. The data base will serve as the common vehicle for further analysis and development.

The behaviors of the specified systems may be automatically displayed by means of a universal interpreter (simulator) for the formal specification itself. Thus the correspondence between the design and its simulation model is ensured.

The present language specification should be viewed as a prototype. We need to implement a version of the design laboratory based on the current language in order to gain necessary design experience. As a result of that experience, we will surely wish to modify and extend the current language.

It is difficult to acquire such experience by hand simulations since the required analyses are designed for computer processing.

The immediate future work must then involve the design and implementation of CAFEL. With the CAFEL we can support experiments designed to validate the formal concepts, to identify deficiencies, and to extend the current techniques of distributed system design.

Upon completion of the above experiments, we may design a production version of CAFEL that would be suitable for general utilization by system engineers and designers.

3.5 Methodology

Our methodology consists of a set of procedures for generating specifications, a set of automatable transformation and analysis tools, and a set of rules for their use. As was discussed in Section 3.1, we will constrain our procedures to support a homogeneous development process starting with very abstract specifications (requirements) and ending with very detailed specifications (hardware and software designs).

In effect, the set of methodology procedures defines a space of development processes. Each development process will consist of some sequence of application of methodology procedures to the results of previous development steps. Each specification will define the current state of a development process.

3.5.1 Introduction

Each methodology procedure must terminate in a specification that is testable for the properties incorporated by that procedure. This implies an automated validation of the results, and is a severe constraint on our methodology.

Each methodology procedure must also be practically applicable to very large specifications. This eliminates many types of analysis and transformation. Fortunately, we can show how to support a large set of powerful development processes based only on procedures that satisfy our requirements.

There are three types of methodology procedures, designer generated, designer guided, and algorithmic procedures.

A designer may simply generate a specification as the next state of the development process. This places a severe burden on the development of automated validation of the resulting specification. Such a step is feasible only in certain special circumstances as discussed below.

A designer may guide an automated generator for the next specification. The required validation may be obtained by either generating only valid results or by generating only results that can be automatically tested and rejected if unsatisfactory.

The methodology procedure may be an algorithm that will simply produce a valid result. A designer could supply parameters to such an algorithm, but validation of the results is assured by the algorithm.

We have identified a set of methodology procedures that are sufficient to support a large set of powerful development processes. These procedures are briefly described in the following subsections. Clearly some experience in their use may lead to their modification and extension in future work.

3.5.2 Approximation

There are two types of approximation procedures, informal and formal. In the former, satisfactory behavior of the

resulting specification as confirmed by the designer is sufficient for development process step completion. In the latter, there is some formally specified correct behavior for comparison with the resulting specification. In both cases, the resulting specification is a formal one that precisely specifies a system that may be an approximation to the desired system.

3.5.2.1 Informal Approximation

A development process step may consist of a designer producing a specification that, upon analysis, is more satisfactory than the previous specification (if any). The required tools are simply those provided in the Computer Assisted Formal Engineering Laboratory described in Section 3.4.

The "more satisfactory" condition is informal and not validated by CAFEL. The engineering laboratory will allow the designer to run whatever simulation experiments or analysis required to conclude that the specification is "more satisfactory."

This type of development step is particularly suitable as an initial step of a development process that will subsequently elaborate and optimize the specification. If this type of step is used later in a development process, CAFEL does not automatically ensure the consistency of the resulting specification with the preceding specification. Thus previous design

decisions may no longer be valid for the new approximation specification.

A development process consisting of only steps of this kind is essentially similar to current practice with the addition of our formal specification techniques for describing design decisions and of significantly greater analysis and simulation tools in CAFEL. The use of CAFEL for this type of development process already constitutes a significant improvement of contemporary design methodologies for large scale systems. The existence of CAFEL itself is sufficient to support such development processes.

3.5.2.2 Formal Approximation

If we develop a formal way to specify required system properties independently of the processes that would produce them, we could provide a CAFEL analysis tool for comparing a specification with those originating requirements. This would be somewhat analogous to validating a proposed implementation of an abstract data type. (If the required properties were axiomatically specified by process independent relations between system attributes we might not be able to generate a suitable metric for ordering approximations.) CAFEL could then compare using that metric and formally validate that a new approximation is closer to the desired system.

When a metric of "correctness" can be established, we can introduce a formal approximation procedure to our methodology. We do not, currently, have such a metric to propose. An alternative to formal approximation steps does exist in the form of elaboration steps described in a subsequent section.

3.5.3 Decomposition/Integration

The complexity and scale of the development processes require some way to decompose them into simpler development processes whose results may be subsequently integrated. We may distinguish two kinds of decomposition, informal and formal. We must consider decomposition and integration procedures together since, in general, how something is to be put together depends on how it was taken apart.

3.5.3.1 Information Decomposition/Integration

If we informally generate several specifications (each of a subset of the required system properties) that jointly specify the entire set of system properties, we could then carry out a development process for each of the specifications. The subsequent integration of the resulting designs must also be informal.

An example of this type of decomposition is to produce separate specifications for site implementations (e.g., power, weight, size, color, etc.), hardware implementations (e.g., TTL logic, CDC 7600, etc.), and software implementations (e.g.,

functions to be performed, responses to be generated, etc.). Each specification could be used to develop designs with some interactions between the development processes. This is a common way to decompose development processes.

The major difficulty with informal decomposition is discovered during the subsequent integration step in which the separate designs must be consistently combined. As long as the decomposition is informal, the composition (integration) must also be informal. It is thus impossible to provide formal analysis tools sufficient to insure that the separately developed specifications will be consistent.

The existence of inconsistencies at system integration time results in very expensive problems. The use of informal decomposition and integration steps will be supported by CAFEL only to the extent that the separate development processes are supported. Interactions between development processes will also be supported by the interactive distributed data base in CAFEL.

If designers wish (or need) to use this type of decomposition, CAFEL will allow them to do so. However, CAFEL cannot ensure the consistency either of the separate initial specifications or of the resulting specifications. CAFEL cannot automatically generate the integrated specifications. It can, of course, analyze and display the behavior of any specification. A designer may thus carry out any desired tests or studies using CAFEL. The existence of CAFEL itself is sufficient for supporting this type of development step.

3.5.3.2 Formal Decomposition/Integration

The difficulties involved in an informal decomposition may be partially overcome if we can find a way to validate the equivalence of the decomposed set of specifications with the original specification. If we also constrain the decomposition by requiring that the resulting designs can be consistently integrated, we would then insure that the development process would work properly (without introduction of errors) if the separate decomposed development processes worked properly. This type of decomposition thus insures the validity of the overall process based on the validity of the separate processes. There may be many different formal decomposition/integration procedures, but the simplest one may be adequate for our purposes and is described below.

The decomposition procedure is to simply copy the original specification and constrain subsequent development of each specification to disjoint subsets of the functions in the specification. Thus each development is carried out in the context of the entire system. All functions not delegated to a particular development process will remain invariant until some subsequent integration. The equivalence of the decomposed specifications with the original specification is trivially ensured by the copy operation.

The separate development processes must make only local design decisions for those functions that can be modified in

that process. Thus, if each development process produces a specification with satisfactory behavior, they may be automatically integrated into a specification that has satisfactory behavior. The integration step consists of constructing a specification by combining the noninvariant functions from each of the separate specifications.

The only possible difficulty with this integration is that the resulting system may have been "overdesigned" since each development must provide proper responses to the original behaviors of the invariant functions in spite of the possibility that another development process may have specialized those functions. The integration step will thus not introduce any new behaviors or errors. It may discover overdesign, but the resulting system still works correctly.

CAFEL can thus provide both formal decomposition and integration procedures that are entirely automated. Decomposition may be carried out in any development state and integration among the component development processes may be done in any subsequent state. Intermediate integration of pairs of component processes can be done at any step with the resulting processes integrated with the remaining processes on some subsequent step. Asymmetric interactions, in which one process integrates results of another, but both continue separately, are also supported by CAFEL. Intermediate integration is a management tool to avoid the over design problem previously mentioned.

Management may decide when and how much to decompose and integrate at each development step without affecting the validity of the development processes. This capability alone will greatly simplify the task of managing a development process while delegating design responsibilities.

3.5.4 Elaboration

An elaboration procedure will transform a specification S into a resulting specification S' such that S is an abstraction of S'. For example, if a primitive function is defined as an expression involving new primitive functions and only local exchange classes, the behavior of the new function will be only a subset of the possible behaviors of the old (primitive) function. Thus an elaboration procedure consists of accepting a new definition for a primitive function or set, and testing for the subsetting of behavior of the old specification. An algorithm for this procedure can be defined and the resulting abstraction relation between S and S' is automatically insured.

The effect of an elaboration step is not to introduce new behavior but rather is to eliminate unneeded behavior. Thus a development process that starts with a very abstract (general) system can progress via elaboration steps to a detailed specification of what must be done. At each step the validity of previous design decisions is preserved and errors caused by violating them cannot be introduced. Thus specifications cannot only be shown to be consistent, but a sequence of specifications

can be shown to be formal abstractions. Invalid elaborations will be detected and prevented by CAFEL.

CAFEL will support elaboration steps and the required analysis may easily be localized to a small portion of the specifications. Thus the sophisticated analysis required is still practical.

3.5.5 Optimization

The intrinsic nature of an optimization procedure is to transform a specification S into an equivalent specification S' such that some objective function of the attributes of a specification will have a better value for S' than for S. A true optimum (i.e., there is no specification S" that will provide a better value) for a complex design is probably neither attainable nor recognizable. We are interested in supporting a designer's efforts to produce a better specification.

There are two problems involved in an optimization step. The first is to evaluate the objective function on a specification. The second is to show that S and S' are equivalent in their behavior.

The formal nature of our specification and their associated attributes for analysis and objective function evaluation. The analysis and simulation tools in CAFEL should be adequate for most practical purposes. Certainly, any practical objective function defined in terms of attributes (formal or informal) of our specifications can be expressed as an analysis procedure in

CAFEL operating on the design data base. Thus CAFEL itself provides a suitable framework in which all possible objective functions may be evaluated.

The second problem of equivalence mentioned above is not so trivially solved. In general, a demonstration of the equivalence, or lack of it, between two arbitrary specifications is impossible. Even with severe constraints that make such an analysis possible, it will usually be impractical. We must find another way. One such way is to find a set of equivalence transformations that when applied to S will produce only equivalent S'. We then constraint our optimization search to systems generated by such transformations. We may still be able to generate more equivalent systems than we can afford. The search techniques to be used may be dependent on the objective function. This problem is quite analogous to that of producing an optimizing compiler and some of that optimization theory may be useful here. In any case the essential thing is to provide a set of powerful and general equivalence transformations.

A study of such equivalence transformations has been carried out and is reported in Appendix C. A general and complete set of "structural equivalence" transformations have been developed.

CAFEL will provide both the objective function evaluations and the equivalence transformations of Appendix C. A designer

may use his skill, experience, or optimization theory in guiding his use of these tools in finding a better specification. CAFEL will ensure that the resulting specification is equivalent (we haven't lost or gained system behaviors) and provides a better objective function value. The decision as to when to stop trying for a still better specification is left to the designer. Again, CAFEL will prevent the introduction of errors in an optimization step.

3.5.6 Evolution

The ordinary computational processes of a system leave that system itself invariant. An evolutionary step of a system is one that transforms the system into a new system. A particularly simple type of evolutionary step is one in which the state of the system is invariant to the step. The initial state (the last state in the old system) of the evolutionary step becomes the initial state for the new system. The only thing that changes in this type of evolution step. We can define evolutionary processes and realize them in system implementations containing generators of the new system. We could design a kernel system that realized a set of evolutionary processes and permit it to evolve as required over its life cycle.

In effect an evolutionary step is a redesign and re-realization in real-time and on-line. Such evolutionary processes

correspond to computations of a "design system" and their realization involves incorporating part of the design system in the system being designed. Thus such design decisions may be deferred until the system being designed is in its operational environment.

Our formal design system is well suited to specify both the evolutionary and computation processes of an application system. The formal nature of our design system makes it possible for many types of evolutionary operations to be realized in application system implementations. We have not studied, as yet, the structures of such realizations since our design theory is focused on the earlier phases of development. It is easy to see that at least some of our methodology steps can be deferred to operational systems.

We may define a "re-elaboration" procedure as part of our methodology. If this procedure can be deferred to operational systems, it would be a powerful evolutionary operator. It is an important procedure for our design methodology since it will enable a designer to change system behavior, in a controlled and controllable way, while responding to changes in requirements, technology, or understanding of solutions.

A re-elaboration step consists of changing some non-primitive sets and functions into primitives. The behavior of the resulting system will thus be a generalization of the initial system. New behavior may be introduced, with the scope

of the new primitives. A re-elaboration step can be followed by a redevelopment with different design decisions

By tying the changes to abstractions and elaborations, we can formally determine the ramifications of those changes and constrain them to be only local if desired. The previous design decisions down to the level of the new primitives will be invariantly maintained and such errors will not be introduced by the changes. The formal behavior of the new specification can, of course, be compared automatically with that of the previous specifications.

CAFEL can provide the automatic analysis of the scope of such changes, validate their locality and preserve more global design decisions if desired. The subsequent redevelopment is supported in the same way as the original development by CAFEL.

3.5.7 Conclusions

We have identified a set of automatable procedures that can, with designer interaction, transform specifications in useful ways. The set of procedures include the following:

-- approximation
-- decomposition/integration
-- elaboration
-- optimization
-- evolution.

Each step can be supported in CAFEL and the validity of the step

insured by analysis tools in CAFEL. Many sources of development error thus become impossible.

The required analysis and transformation procedures can be practically carried out even on large scale specifications. The validation and design feedback generated by CAFEL will allow the designer to concentrate on analysis and solution to critical design problems. The resulting design decisions can be introduced into the specifications by sequences of the above methodology steps. The computer becomes an active partner in the development process but does not dictate the design decisions. A poor designer may still produce a poor design. But if he does, it is possible to analyze and compare it with other designs prior to implementation.

The methodology procedures above are more than sufficient to support conventional development steps. In addition, many formal development steps are possible with automatic validation of the changes made in that step.

Future work must include the detailed development of algorithms for the above procedures, of variants of those procedures, and identification of potentially useful new kinds of procedures.

3.6 Design Principles

We have done little work on this area since all such principles must be tested by practical experience to gain credence. Most such principles will arise from practical experience. We must complete a design and implementation of CAFEL in order to gain the required experience. Clearly, our formal methodology will not prevent (and may provide substantial support for) using any design principles found effective. CAFEL provides a methodology for carrying out design not an enforcement of a particular design process. We can, however, identify some general principles that can be plausibly justified a priori. They must still be tested by experience.

3.6.1 Formal Methodology

The first principle is to maximize the use of a formal methodology and to minimize introduction of informal steps in a formal development process.

This principle can be justified on the basis that maximal use of the formal methodology will maximize design automation, testability of specifications, traceability of design decisions, and early detection of errors while minimizing or eliminating many types of development errors.

Informal steps that produce specifications not formally related to the initial specifications introduce discontinuities in the step validation and prevent automatic assurance of

previous design decision invariances. Many types of new errors could be undetectably introduced and all previous analysis would have to be repeated. All relationships between previous and the new specifications must be established by analysis. Unfortunately, such analysis is frequently impossible since arbitrarily difficult worst case conditions may arise. Within the formal methodology, such worst case conditions are carefully avoided by generating the new specification from the old. Such steps correspond to approximation steps in our methodology and can be supported as such by CAFEL.

The current method of using different specification languages in each development phase and informally (by hand) generating "equivalent" output in the language of the next phase is a gross violation of this principle and severely handicaps a design validation program.

The current methods also use informal specifications (that are quite imprecise) as approximations to the desired system. The amount of automated analysis that can be done on such specifications is quite limited in comparison to those provided by CAFEL.

A corollary to this principle is that the initial approximation step should be as abstract as is possible. This provides maximum scope for a formal methodology designed to produce less abstract specifications.

Another corollary is that the final formal specification should be sufficiently detailed to be input to an automated implementation methodology. We should not leave the formal for the informal (realization) until required, and we should generate the realization using tested and validated tools. The problem of validating an implementation is otherwise very difficult and expensive.

3.6.2 Closure

We can distinguish between open and closed system design. An open system is a subsystem with exchange functions in some inter-subsystem class. The external exchange functions that complete the exchange class are left unspecified. The one-sided specification of such external interactions is equivalent to a conventional interface specification. The behavior of an open system specification may only be formally analyzed with respect to all possible external designs (closures) and all subsystem design decisions must be satisfactory for any closure. These are significant design constraints.

A closed system has no external interactions. For example, a control system and the things being controlled may form a closed system and all behavior of the control system may be defined in terms of behaviors in the controlled systems. Our formal specification methodology will support the specification of both control and controlling systems.

The normal decomposition method is to partition a system into open subsystems and design each separately until system integration time. At that time, the inconsistencies, errors, and over-design are discovered. In the meantime, each subsystem designer has had to produce some environment simulation driver to test his designs. The design of these environments is similar to the design of the controllers and is fraught with the same problems. Inconsistencies in these designs can lead to serious development errors that are detected no earlier than system integration. These difficulties suggest a design closure principle.

The design closure principle is that system specifications should be closed and formal decompositions (rather than partitions) should be used to factor development processes.

The benefits of using closed system specifications are many. We not only provide automatically consistent environment models to each decomposed development process, but also provide an easy integration technique for absorbing improved models as the development processes proceed.

In particular, it should be noted that, using closed system specifications, the design of the processes to be controlled (the problem space) and the design of the controlling processes (the solution space) can be carried out jointly at each level of abstraction. Control system behavior can be

studied in terms of controlled process behavior. A customer may then analyze simulation results provided by CAFEL entirely in terms of his problem space, independent of an understanding of the details of the solution processes. A close formal relationship can thus be established between originating requirements and proposed closed system specifications. CAFEL can provide complete support for such analysis and simulation.

3.6.3 Unbounded Resources

The performance evaluation of a system specification is a severe problem. The full complexity of a design problem arises from attempts to optimize performance. If we are given an unbounded amount of resources for use by the system realization, the subsequent optimization can deal with the resource constraints as discussed in the next section.

The unbounded resource principle is that we should first design (to a testable level of abstraction) without realization resource constraints until we have a satisfactory design in all other respects.

At the end of the above design we will have a valid system that may be overdesigned in terms of performance. Because of the comparative simplicity of the design it will be much easier to validate the functional relationships and predict the performance. If the system does not overperform, we know we must redesign immediately. Further investment in complex

optimization could not produce better results. Thus we know at this point whether the system could be realized and meet the logical and performance requirements. It might of course be much too expensive, etc., to actually build in this form. A subsequent optimization phase may be required.

We have thus simplified this phase of design and obtained a valid design. This design may be used as the input to the various equivalence transformations useful for optimization.

CAFEL will support this type of development process as a sequence of approximation and elaboration steps with full analysis of each specification including performance.

3.6.4 Optimization

If we have a valid design that overperforms we may optimize by finding an equivalent design in which the overperformance has been degraded. This may easily (?) be done by introducing resource contention by sharing resources used by overperforming functions.

First we can use CAFEL to identify the overperforming functions and the redundant resources causing them to overperform. We need not guess, since the analysis of an initial unbounded resource specification is relatively easy to automate in CAFEL.

Second, we can degrade (and minimize resource cost) performance by sharing resources and decreasing resource

redundancy. This step consists of an equivalence transformation to a new specification with contention (and sharing) of minimized resources. The equivalency transformations will be automatic and provided by CAFEL. The contention resolution mechanisms introduced will be constrained to those for which a lower bound on resulting performance can be evaluated.

Third, we can analyze the resulting specifications to ensure that the requirements are still being met. This is a potentially complex analysis and more research on contention models (operating system theory) may be required. We do have control over the complexity (at the price of resource waste) of the contention models introduced above. Thus we can produce an optimized system without introducing errors and knowing if performance requirements will be met.

The resulting specifications will be far more complex than the unbounded resource specifications input to the optimization phase. For example, evaluation path performances are coupled (even if otherwise completely independent) via contention for shared resources. Since the sharing of resources may be on a global basis, it is no longer even possible to do performance analysis locally for each function. What used to be local functional elaborations (and locally testable) may be turned "inside out" by global interactions with shared resources.

We sorely need a theory of operating systems that will allow us to improve our resulting utilization of resources and

still to predict lower performance bounds. It does us little good if our overperforming unbounded resource specification is so optimized that it fails to meet the performance requirements. By following the above principles, we can produce valid systems that will be testable for performance requirements. They may still be too expensive at the current level of operating systems theory. But they will work. Consider the alternatives.

The recent experience with technology and the analysis of life cycle costs suggest that some waste of realization resources may be very cost effective. The resulting simplicity may produce far greater payoffs in the development process.

A further interesting question is when and how often to optimize the design. A possible solution would be to optimize at the end of each phase to demonstrate feasibility of the design, then to "throw away" the optimized specification, and start the next phase with the unoptimized design. This will lead to a simpler development process and a maximally global optimization just prior to implementation.

There may be useful hierarchies of resources such that optimization with respect to the resources of the current phase can be carried out and preserved through subsequent phases. This requires the establishment of a theorem similar to those underlying the current state of compiler optimization theory. This would be an interesting avenue for further work.

We must gain experience in the use of CAFEL to validate this optimization principle. The potential payoffs are enormous. But only experiments can assess the validity of this principle.

3.6.5 Evolutionary Adaptation

Hindsight is always better than foresight. In a world of rapidly changing requirements and technology, hindsight may be the only insight available. A logical extension of this fact suggests the following principle.

At each step of development, make only those design decisions that are required in order to delegate all the rest of the design decisions to subsequent steps.

The consequences of this principle are to preserve maximum freedom of re-elaboration changes with minimum scope of consequences. In the limit, the principle requires maximal exploitation of evolutionary operators in the deployed system. Hindsight during operation may then be employed in a practical way to faciliate adaptation. This allows earlier deployment and greater adaptability to change.

CAFEL and the associated formal methodology will greatly improve our ability to defer design decisions even until operational status due both to the precision of constraining and specifying those evolutionary changes and to the automatability of the "run time" redesign and re-realizations made possible by the formal methodology.

3.6.6 Conclusions

The area of design principles to be used in selecting sequences of formal methodology steps and in guiding design decisions is a fruitful and little explored area. A major advantage of a formal design methodology is that it provides a theoretical and practical framework within which precise principles may be developed and validated.

The current structure of our formal design theory is clearly adequate to formulate and validate a large variety of potentially valuable design principles. The generality of our theory makes such a research and experiment investment worthwhile since they can be applied to a vast domain of distributed system developments and to each phase as well.

CAFEL provides an excellent vehicle for capturing such design experience in formal principles, methods, and specifications. CAFEL itself is designed to be highly evolutionary so as to incorporate subsequent design experience as an incremental improvement of our design theory.

Given such experience the next work on the design theory would consist of developing a suitable operating system theory and formally incorporating performance properties in our formal specifications. In other words, it would then be time to go around our research procedure one more time while experience with the presently conceived CAFEL guided and "paid" the way.

It is clear that CAFEL, even as presently conceived, would significantly improve the current state of design theory and practice. A demonstration of this is, of course, required for creditability but must await a prototype implementation of CAFEL and the design of suitable development experiments.

4. Patient Monitoring Example

This section represents the major portion of the paper "The Use of Formal Asynchronous Process Specifications in a System Development Process" by D. R. Fitzwater and Pamela Zave, presented at the Texas Conference on Computing, Fall 1977.

4.1 A Development Process

A spectrum of design methods and a start on a design theory, all based on these formal specifications, have been developed [FD]. This report is intended to illustrate the use of these specifications in development without aiming towards a particular development process. They are aimed towards a more formal development process that can be substantially supported with automatic design tools and analysis. Each specification can be effectively tested for the behavior of its specified system as well as for completeness and consistency.

A major point to notice is the distinction between informal (vague) specifications and formal (precise) specifications. We can provide few design and analysis tools applicable to informal specifications since information content is not automatically analyzable. We must, therefore, introduce our formal specifications at the earliest possible point in development.

At early stages, only approximate specifications can be given. This does not imply that the specifications must be informal. We can instead give a formal specification of a precise system whose behavior approximates the desired system.

[FD] FITZWATER, D. R. "The Formal Design and Analysis of Distributed Data-Processing Systems." Computer Sciences Technical Report 295, March 1977. Department of Computer Sciences, University of Wisconsin, Madison, Wisconsin.

We can then study the degree of approximation by analysis of the specification. This is not feasible with informal specifications.

The level of abstraction of our specifications is entirely dependent on the primitive functions and sets used in its definition. We assume that any development process will start with very abstract primitives and end with detailed realizations. The following examples are thus ordered by their expected sequence of occurrence in a development process. They are not unique, and need not occur in any given development.

For convenience and economy of presentation, the definition sets that constitute a specification are also ordered. All definitions hold for subsequent definition sets unless explicitly redefined. Thus definitions will usually appear only once even if they are used in several specifications.

The design problem used for the examples is quoted from [SMC] as (statement numbers are added):

> "1. A patient monitoring system is required for a hospital.
>
> 2. Each patient is monitored by an analog device which measures factors such as pulse, temperature, blood pressure, and skin resistance.

[SMC] STEVENS, W. P., Myers, G. F., and Constantine, L. C. "Structured Design." IBM Systems Journal 13, No. 2, 1974.

> 3. The program reads these factors on a periodic basis (specified for each patient) and stores these factors in a data base.
>
> 4. For each patient, safe ranges for each factor are specified (e.g., patient X's valid temperature range is 98 to 99.5 degrees Fahrenheit).
>
> 5. If a factor falls outside of the patient's safe range, or if an analog device fails, the nurse's station is notified."

Since a customer for our design is not present to validate our assumptions as to what behavior is required in detail, we will make as few as possible and keep our development to a high level of abstraction. From these examples it should be obvious that we could elaborate the specifications to arbitrarily low level primitives. In particular we will stop prior to deciding which functions will be hardware and which will be software.

We will also start with the assumption that processes and functions are not scarce resources and produce specifications that utilize maximum resources. A performance test at that point will decide if there is any point to continuing with that specification (i.e., does it meet performance requirements?). If it does, we could then share processes or function evaluations to decrease performance where allowed. Preliminary results with dynamic analysis strongly indicate that this is a good development plan. Any sharing of resources for optimization purposes will complicate dynamic forecasting

because it couples previously independent computation paths via resource contention. The design should be made correctly and tested prior to such optimization.

4.2 Idealized Behavior

The first example is not one of our specifications. It is an interface characterization of idealized behavior consistent with the original problem. We will use this example for comparison with our process specifications.

Figure 4.2-1 defines, as equations, the relations between values and set members in the sets patient, data-base, range-space, and nurse-station-space. The defined values are not time dependent and no process of evaluation is specified. These equations are formal definitions of the relationships described by the problem. Note that questions of performance are irrelevant, since nothing is "done."

If the problem were much more complex, the sets and equations could become too complex to generate, and too lengthy to be practical. There are in general no algorithms for deciding whether a system of equations is complete or consistent and the equations involved might not be solvable to yield information about the specified system.

This interface definition is more abstract than subsequent process specifications and cannot be exactly realized since no time lag is allowed between patient factor values and the resulting notify-status value. We could interpret such equations as idealized (but not sufficient) constraints on further design and can compare the behavior of the system

$\forall P\text{-}F \in$ DATE-BASE, & $\forall$ RANGE $\in$ RANGE-SPACE,

PATIENT-MONITOR (P-F,RANGE) = $(i,\text{TIME},\bigvee_{0\le j\le k_2} LB_{ij} > P\text{-}F_{ij} \vee P\text{-}F_{ij} > HB_{ij})$

where P-F $\equiv (i,\text{TIME},(PF_{i1},\ldots,PF_{ik_2}))$

RANGE $\equiv (i,((LB_{i1},HB_{i1}),\ldots,(LB_{ik_2},HB_{ik_2}))$

$\forall(i,\text{TIME},\text{STATUS}) \in$ PATIENTS,ANALOG-DEVICE $((i,\text{TIME},\text{STATUS})) = (i,\text{TIME},(PF_{i1},\ldots,PF_{ik_2}))$

"PF_{ij} are implicitly defined in terms of patient status."

PATIENT-MONITOR: DATA-BASE × RANGE-SPACE → NURSE-STATION-SPACE

"defines notifications."

ANALOG-DEVICE: PATIENTS → DATA-BASE

"is primitive function defining patient i factor values from status at time."

DATA-BASE $\equiv$ ID × TIMES × FACTOR-SPACE "patient i has factors at times."

FACTOR-SPACE $\equiv \prod_{0\le j\le k_2}$ FACTOR "factors are common to all patients. Factor F_0 is device status."

RANGE-SPACE $\equiv$ ID × BOUNDS "bounds are local for each patient."

BOUNDS $\equiv \prod_{0\le j\le k_2}$ FACTOR-PAIR "defines ranges"

FACTOR-PAIR $\equiv$ FACTOR × FACTOR "element is (low-bound,high-bound)"

PATIENTS $\equiv$ ID × TIMES × STATUS-SPACE "defines observable patients"

NURSE-STATION-SPACE $\equiv$ ID × TIMES × NOTIFY-STATUS

NOTIFY-STATUS $\equiv \prod_{0\le j\le k_2}$ B "an out of range Boolean tuple"

ID $\equiv \{1,\ldots,k_1\}$ "patient identification"

TIMES $\in N_{k_3}$ "times at which associated status is defined"

B $\equiv \{T,F\}$ "T,F are Boolean values true, false"

FACTOR $\in N_{k_3}$ "values of patient factors"

$k_1 \in N_{k_3}$ "parameter is number of patients"

STATUS-SPACE "patient status is primitive set"

$k_2 \in N_{k_3}$ "parameter is number of factors"

$k_3 \in$ N "bound on integer space"

$N_{k_3} \equiv \{0,1,\ldots,k_3\}$ "bounded integers"

N $\equiv \{0,1,\ldots\}$ "integer space"

Figure 4.2-1. Definition Set 1: Idealized Patient Monitoring Interface Equations.

(when effectively specified as in following examples) against these constraints.

Substantial elaboration of these idealized equations is not attractive since they are not effective and cannot, in general, be used to generate observable behavior for the customer. The essence of the early development process is to specify precisely successive system approximations that can be tested to see if their behavior satisfies the customer for the system. At that point we can then elaborate the specifications to produce a detailed design.

4.3 Process Approximation

The first system specification has very abstract processes whose behavior approximates the interface functions in the previous section. We will first discuss this specification in (possibly redundant) detail as a specification example and then discuss it as a patient monitoring system.

The system contains k_1 patient processes and k_1 monitor processes. The only interactions are between fixed pairs (of same subscript values) of $patient_i$ and $monitor_i$ processes. The $patient_i$ only executes XSM_i (M_i represents the class subscript) exchanges and computes a sequence of status-space values. The patient does not synchronize with the monitor (i.e., if monitor is too slow, patient continues to change). The monitor must synchronize with the patient to receive patient factors (i.e., patient status may not be well defined at a given instant in time). The XSM_i is executived for its side effects only, the P_1^2 projection function discards any XSM_i value. The analog device function transforms the primitive status information into a tuple of factor values.

The state space of the monitor process has five components. The first is an integer that represents the number of process steps until the next measurement of the associated patient process. The second defines the valid ranges of factors. The third is a bounded stack to hold the previous k_4 observations.

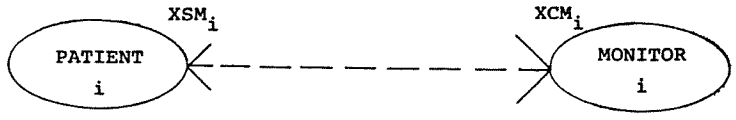


Figure 4.3-1. Patient-Monitoring System. This is an exchange graph of a high level specification that approximates the idealized interface in Definition Set 1. This figure represents $2k_1$ processes.

$PATIENT\text{-}MONITORING\text{-}SYSTEM_1 \equiv (PATIENT_1, MONITOR_1, \ldots, PATIENT_{k_1}, MONITOR_{k_1})$

$PATIENT_i \equiv (PATIENTF_i, STATUS\text{-}SPACE)$ "is process specifying patient i"

$PATIENTF_i$ (STATUS) $\equiv P_1^2$ (PSIM (STATUS, XSM_i(ANALOG-DEVICE(STATUS)))

PSIM: STATUS-SPACE → STATUS-SPACE "generates change in patient status"

$MONITOR_i \equiv (MONITORF_i, MONITORS)$ "is process monitoring patient i"

$MONITORS \equiv N_{k_3} \times RANGE\text{-}SPACE \times STACKS \times PARSET \times NOTIFY\text{-}STATUS$

$MONITORF_i$ (TIME,RANGE,DATA,PAR,NOTICE) $\equiv$ [NEQ(TIME,0): (DECR(TIME),RANGE,DATA,PAR,NOTICE),
T: MON_i(XCM_i(F),RANGE,DATA,PAR,NOTICE)]

MON_i(FACTORS,RANGE,DATA,PAR,NOTICE) $\equiv$ (SCHED(FACTORS,RANGE,DATA,PAR,NOTICE),RANGE,
PUSH(FACTORS,DATA),PAR,OUT-OF-RANGE(FACTORS,RANGE))

P_k^j: $\prod_{1 \le n \le j} Z_n \to Z_k$ "projects j-tuple onto k element, $k \le j$.
Z can be any set"

ANALOG-DEVICE: STATUS-SPACE → FACTOR-SPACE "does A-D conversion"

NEQ: $N_{k_3} \times N_{k_3} \to B$ "predicate on inequality"

DECR: $N_{k_3} \to N_{k_3}$ "integer decrement"

SCHED: FACTOR-SPACE × RANGE-SPACE × STACKS × PARSET × NOTIFY-STATUS → N_{k_3}
"generates time interval for next measurement"

OUT-OF-RANGE: FACTOR-SPACE × RANGE-SPACE → NOTIFY-STATUS "tests patient factors for range violations"

PUSH: FACTOR-SPACE × STACKS → STACKS
"pushes first argument onto k_4 bounded stack of second argument. If bound is reached, bottom element of stack is deleted."

$k_4 \in N_{k_3}$ "is bound on size of stacks"

$k_5 \in N_{k_3}$ "is size of patient status-space element"

$k_6 \in N_{k_3}$ "is size of parset"

$PARSET \equiv \prod_{1 \le j \le k_6} N_{k_3}$ "is scheduling information"

$RANGE\text{-}SPACE \equiv BOUNDS$

$STATUS\text{-}SPACE \equiv \prod_{1 \le j \le k_5} N_{k_3}$ "is status vector"

XSM_i: FACTOR-SPACE → {F}

XCM_i: {F} → FACTOR-SPACE

$STACKS \equiv \prod_{1 \le i \le k_4} N_{k_3}$ "is stack of size $\le k_4$ for data base"

Figure 4.3-2. Definition Set 2: Patient-Monitoring $System_1$.

The fourth is a set of parameters to be used in scheduling. The last is a Boolean vector defining the out-of-range status of the previously measured factors.

The monitor process successor function simply steps and counts down the current interval time once each step until zero. When time is zero, the MON_i function is evaluated after its first argument, XCM_i, is evaluated to receive new patient factors measurements. The MON function generates a new interval time via SCHED, pushes the new measurements onto the local data base stack, and updates notice via out-of-range. The process then takes another step.

In order to produce a specification (completeness and consistency is required) a few assumptions had to be made. These should have been checked out with the customer. We have assumed that the sizes of all numbers (represented as integers), the size of the data base, the size of an element of status-space, and the number of patients are all bounded by parameters. We have assumed that an analog device is local to each patient, can measure each patient state, and reports its own status as factor zero. The parameterized scheduler is the same for all patients although the parameter values need not be. We also assume that saving the notifications from only the latest measurement is sufficient. If any of these assumptions are invalid, the specifications should be changed.

Although there is little detail in this specification and few design decisions have been made, we can simulate the specified system automatically by giving either standard stochastic models or simulating procedures for the primitive functions. The designer or customer can then do experiments and observe specified system behavior from the point of view of the customer.

Performance requirements are now meaningful. How long can a patient process step be and still sufficiently model the patient? The analog device must be as fast. How long can a monitor process step be and still sample often enough? Monitor steps must be of uniform length since they are self-timing. This specification can only be considered an approximation to the interface specification since it can only approach the axiomatic behavior in the limit of infinite performance.

Let us suppose that the designer and the customer come to the following, not very surprising, conclusions. The designer doesn't want monitor to be self-clocking since he anticipates difficulties in making steps uniform in length. The customer agrees with the stated assumptions and notices that patient status-space values are not affected by performance of monitor. He asks to expand design to include feedback from nurse-station via a "nurse" to the patient. The customer also adds that the nurse-station must run independently and that a data-base system already exists for other purposes. The new monitor system should use it to save information.

The results of these conclusions are given as the next example specification.

4.4 Closed Loop System

The second system specification is a better approximation to what the customer wants. An independent (but shared) clock has been added. Both the data base and the nurse-station have been factored into independent processes.

The data-base$_i$ process executes only XSB$_i$ exchange functions and does not synchronize with the monitor process. It may be run freely as desired. The data-base process need model only that aspect of the data-base system that is currently relevant to our patient-monitoring system. The design of this system can thus be decoupled from the data base system design, while ensuring that subsequent integration will not introduce new problems.

The clock process does not synchronize and may easily be implemented in constant length steps since interactions are always immediate, if at all. A process may "read" the clock by executing the XACK exchange function. Only one process may read the clock at a time since it was specified that way.

The nurse-station is specified as a free running process that does not synchronize to interact and executes only XS exchange functions. The nurse-station$_i$ simply buffers the latest out-of-range information for patient i and passes it when requested by nurse$_i$.

Nurse$_i$ synchronizes with nurse-station$_i$ to obtain notify-status values, and must synchronize with patient$_i$ to pass along

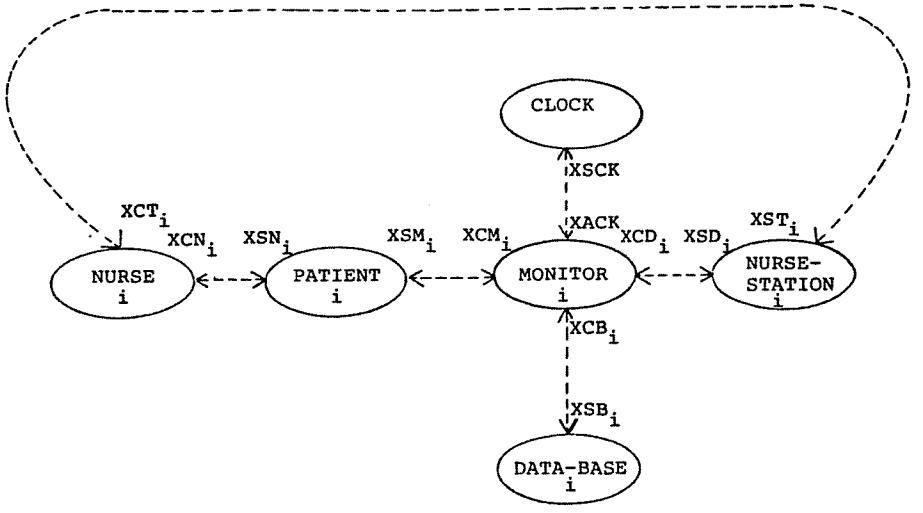


Figure 4.4-1. Patient-Monitoring-System$_2$. This is an exchange graph of a decomposition of previous system with addition of feedback via nurse. This figure represents 5k$_i$+1 processes.

PATIENT-MONITORING-SYSTEM$_2$ ≡ (CLOCK,NURSE$_1$,PATIENT$_1$,MONITOR$_1$,DATA-BASE$_1$,
NURSE-STATION$_1$,...,NURSE$_{k_1}$,PATIENT$_{k_1}$,MONITOR$_{k_1}$,DATA-BASE$_{k_1}$,NURSE-STATION$_{k_1}$)

CLOCK ≡ (CLOCKF,N_{k_3}) "is shared timer process for all processes."

CLOCK(TIME) ≡ F_1^2 (NEWTIME,XCCK(NEWTIME))
where NEWTIME ≡ SUC^1(TIME)

SUC: $N_{k_3} \rightarrow N_{k_3}$

SUC(X) ≡ [EQ(X,k_3): 0,T: ADDONE(X)]

NURSE$_i$ ≡ (NURSEF$_i$,NURSES) "is process simulating nurse$_i$."

NURSES ≡ $\prod_{1 \le i \le k_7} N_{k_3}$

NURSEF$_i$(STATUS) ≡ F_1^2 (NSIM(STATUS,NOTICE),XCN$_i$(TREAT(STATUS,NOTICE)))
where NOTICE ≡ XCT$_i^1$(F)

PATIENT$_i$ ≡ (PATIENTF$_i$,STATUS-SPACE) "is process simulating patient$_i$."

PATIENTF$_i$(STATUS) ≡ F_1^2 (PSIM(STATUS,XSN$_i$(F)),XSM$_i$(ANALOG-DEVICE(STATUS)))

MONITOR$_i$ ≡ (MONITORF$_i$,MONITORS) "is PATIENT$_i$ monitoring process."

MONITORS ≡ N_{k_3} × RANGE-SPACE × PARSET

MONITORF$_i$(TIME,RANGE,PAR) ≡ [GRT(TIME,XACK^1(F)): (TIME,RANGE,PAR),
T: MON$_i$(XACK^1(F),RANGE,PAR,XCM$_i$(F))]

MON$_i$(TIME,RANGE,PAR,FACTORS) ≡ (SCHED(TIME,RANGE,PAR,FACTORS),
F_1^2(RANGE,XCB$_i$((TIME,FACTORS))),
F_1^2(PAR,XCD$_i$(OUT-OF-RANGE(FACTORS,RANGE))))

DATA-BASE$_i$ ≡ (DATA-BASEF$_i$,STACKS) "this process models patient$_i$ data-base."

DATE-BASEF$_i$(DATA) ≡ [NOMSG(MSG): DATA,T: PUSH(MSG,DATA)]
where MSG ≡ XSB$_i^1$(F)

NURSE-STATION$_i$ ≡ (NURSE-STATIONF$_i$,NOTIFY-STATUS) "is process modeling patient$_i$ nurse-station."

NURSE-STATIONF$_i$(NOTICES) ≡ F_1^2 ([NOMSG(MSG): NOTICES,T: MSG], XST$_i$(NOTICES))
where MSG ≡ XSD$_i^1$(F)

SCHED: N_{k_3} × RANGE-SPACE × PARSET × FACTOR-SPACE → N_{k_3} "generates time for next measurement."

NSIM: NURSES × NOTIFY-STATUS → NURSES "simulates nurse state change."

TREAT: NURSES × NOTIFY-STATUS → TMT "generates patient treatment."

ADDONE: $N_{k_3} \rightarrow N_{k_3}$ "integer successor function."

EQ: $N_{k_3} \times N_{k_3} \rightarrow$ B "equality predicate."

GRT: $N_{k_3} \times N_{k_3} \rightarrow$ B "greater-than predicate."

XST$_i$: NOTIFY-STATUS → {F}

XCN$_i$: TMT → {F}

XCB$_i$: N_{i_3} × FACTORS-SPACE → {F}

XSCK: $N_{k_3} \rightarrow$ {F}

XCD$_i$: NOTIFY-STATUS → {F}

NOMSG: Z → B "NOMSG(Y) ≡ if Y = F then T else F"

$k_7 \in N_{k_3}$ "bound on nurse space element."

TMT ≡ N_{k_3} "specifies patient treatment."

XCT$_i$: {F} → NOTIFY-STATUS

XSN$_i$: {F} → TMT

XSB$_i$: {F} → N_{k_3} × FACTOR-SPACE

XACK: {F} → N_{k_3}

XSD$_i$: {F} → NOTIFY-STATUS

Figure 4.4-2. Definition Set 3: Patient-Monitoring-System$_2$.

a TMT value that specifies the desired patient$_i$ treatment.

Patient$_i$ is free running and does not synchronize any interactions but may interact on any step. Patient status is simulated and interactions are left to the "initiative" of other processes.

Monitor$_i$ must synchronize with all of the processes it interacts with, and thus an interacting step length may be greater than or equal to the maximum of the step lengths of the other processes.

The remaining changes are relatively trivial adjustments to the ones described below.

This specification can again be analyzed and the specified system simulated with whatever experiments the customer or designer may care to define. There are many additional aspects the customer or the designer may notice as deficiencies, even at this abstract level, that may require new specifications. The current specification is complete and consistent. It is the customer's business to decide whether he wants the specified system, whose behavior he can study as he wishes.

We are here interested in illustrating a succession of development steps rather than producing a fully elaborated design. We will consequently assume that the customer is happy with the behavior of specified system to the extent that it is elaborated. He does believe that there should be fewer nurses than patients. The validity of that belief cannot be

tested by analysis of our specifications without more elaboration of the nurse and patient processes. We will accept it for now.

The designer, using the performance estimates obtained as answers to the questions of the previous specification, now decides that he does not need multiple monitor, nurse-station, and data-base processes and that he can save resources by multiplexing them.

We are thus led, by minimizing performance where it is too good, to the more optimum, shared process model in the next specification. Until this point, there has been no performance coupling between processes associated with different patients and performance analysis has been comparatively simple.

4.5 Process Sharing

When we consider processes to be a scarce resource and share them, previously independent design decisions and performances now become coupled and we must ensure conflict resolution in the shared processes.

The $patient_i$ and clock processes remain the same. We must introduce buffer processes to save notices and introduce minor adjustments in the other processes. Only monitor, data-base, and nurse-station require significant changes. Nurses are modified only to resolve the contention for patients and notifications.

The interactiions of monitor remain the same, except that the data-base message must now include patient identification. Synchronizations are the same as before. SCHED must produce not only the time of the next measurement, but also identification of which patient is to be measured.

The data-base operates just as before except that the state is a k_1-tuple of stacks and that only the selected one is updated.

The $buffer_i$ processes are the residuals of the old nurse-station processes and simply buffer the out-of-range notices without synchronization. Old notices are overwritten; newest notices are passed to nurse-station.

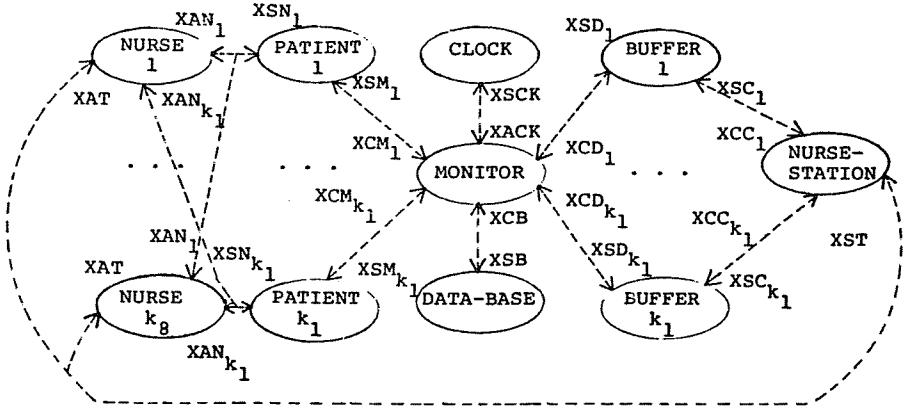


Figure 4.5-1. Patient-Monitoring-System$_3$. This is an exchange graph of a shared process version of the previous specification. This figure represents $2k_1 + k_8 + 4$ processes.

PATIENT-MONITORING-SYSTEM$_3$ ≡ (CLOCK,MONITOR,DATA-BASE,NURSE-STATION,

NURSE$_1$,..,NURSE$_{k_8}$,PATIENT$_1$,BUFFER$_1$,...,PATIENT$_{k_1}$,BUFFER$_{k_1}$)

MONITOR ≡ (MONITORF,MONITORS) "is process that monitors all patients."

MONITORS ≡ NEXT-OBS × RANGE-SPACES × PARSETS

NEXT-OBS ≡ $N_{k_3} \times F_{k_3}$ "(time,i) defines time and patient for next measurement."

MONITORF((TIME,i),RANGE,PAR) ≡ [GET(TIME,$XACK^1$(F)): ((TIME,i),RANGE,PAR),

T: MON($XACK^1$(F),i,RANGE,PAR,[EQ(i,1): XCM_1(F),EQ(i,2):XCM_2(F),..,T: XCM_{k_1} (F))]]

MON(TIME,i,RANGE,PAR,FACTORS) ≡ (SCHED((TIME,i),RANGE,PAR,FACTORS),

P_1^2 (RANGE,XCB((TIME,i),FACTORS))), P_1^2 (PAR,[EQ(i,1):Y_1,..,T:Y_{k_1}]))

where Y_i ≡ XCD_i(OUT-OF-RANGE(FACTORS,P_i^1 (RANGE)))

DATA-BASE ≡ (DATA-BASEF,DATA-BASES) "is process that stores data for all patients."

DATA-BASES ≡ $\prod_{1 \le j \le k_1}$ STACK "is k_1-tuple of all stacks."

DATA-BASEF(S_1,..,S_{k_1}) ≡ [NOMSG(MSG):(S_1,..,S_{k_1}),T: SAVE(MSG,S_1,...,S_{k_1})]

where MSG ≡ XSB^1(F)

SAVE(((TIME,i),FACTORS),S_1,..,S_{k_1}) ≡ [EQ(i,1):(Y_1,S_2,..,S_{k_1}),..,T:(S_1,S_2,...,Y_{k_1})]

where Y_i ≡ PUSH((TIME,FACTORS),S_i)

$BUFFER_i$ ≡ ($BUFFERF_i$,NOTIFY-STATUS) "is process holding previous notices."

$BUFFERF_i$(NOTICE) ≡ BUF_i([NOMSG(MSG): NOTICE,T:MSG])

where MSG ≡ XSD_i^1(F)

BUF_i(NOTICE) ≡ P_1^2 (NOTICE,XSC_i(NOTICE))

NURSE-STATION ≡ (NURSE-STATIONF,N_{k_3}) "process 'displays' notice on patient at a time in a cycle."

NURSE-STATIONF(j) ≡ P_1^2 ([EQ(j,k_1): 1,T: SUC(j)],

XST((j,[EQ(j,1): XCC_1(F),..,T:XCC_{k_1} (F)])))

$NURSE_i$ ≡ ($NURSEF_i$,NURSES) "process simulates $nurse_i$."

$NURSEF_i$(STATUS) ≡ P_i^2 (NSIM(STATUS,Y), [EQ(j,1): XAN_1(Z),..,T:XAN_{k_1} (Z)])

where Y ≡ XAT^1(F), j ≡ P_1^2 (Y), Z ≡ TREAT (STATUS, Y)

SCHED: NEXT-OBS × RANGE-SPACES × PARSETS × FACTOR-SPACE × NEXT-OBS

"generates patient ID and time for next measurement."

NSIM: NURSES × NOTIFY-STATUS$_i$ → NURSES "simulate nurse change of state."

NOTIFY-STATUS$_i$ ≡ ID × NOTIFY-STATUS

XST: NOTIFY-STATUS$_i$ → {F} XCT:{F} → NOTIFY-STATUS

XAN_i: TXT → {F} XSN_i:{F} → TXT

XCB: NEXT-OBS × FACTOR-SPACE → {F} XSB:{F} → NEXT-OBS × FACTOR-SPACE

XSC_i: NOTIFY-STATUS → {F} XCC_i:{F} → NOTIFY-STATUS

$k_8 \in N_{k_3}$ "is number of nurses."

RANGE-SPACES ≡ $\prod_{1 \le i \le k_1}$ RANGE-SPACE "k_1-tuple of all ranges."

PARSETS ≡ $\prod_{1 \le j \le k_1}$ PARSET "k_1-tuple of all parameters."

Figure 4.5-2. Definition Set 4: Patient-Monitoring-System$_3$.

The new nurse-station process now must cycle through the buffer processes, displaying each notice in turn, without synchronizing with any nurse who might see the notice. Each notice will be delivered to at most one nurse.

All of the nurses share the nurse-station and compete for notices. Only one will receive a given notice that identifies the corresponding patient. That nurse then specifies treatment for the identified patient.

This sytem specification can now be analyzed and simulated by both the customer and the designer. There are now some nontrivial and interesting performance questions to be resolved. The important thing to notice is that they can be observed and resolved, even at this high level of abstraction, since all nonlocal interactions are specified. There are a number of issues likely to be discovered by such an analysis and simulation and their resolution will probably involve producing new specifications. We will assume, in the interest of getting on with our examples, that no such changes are required.

The next step will be to elaborate our specifications by making some further design assumptions.

4.6 Process Elaboration

We have so far carefully selected our primitive functions so that only local interactions would be required in their elaborations. We can thus do such elaboration without destroying either the previous specifications or the results of their analysis. Elaboration may place further constraints on previous behavior and make more detailed analysis and simulation possible.

We have not been given sufficient information, as yet, to specify a scheduling policy.

We will assume (subject to customer approval of the resulting behavior) a linear scheduling policy. The scheduled time, NEXT, for the next measurement of $patient_i$ will be given by the sum of the previous measurement time and a constraint DEL_i given for each patient.

The linear assumption implies changes to the function MONITORF without changing previous process interactions. A new definition of MONITORF is given in definition set 5 (Fig. 4.6-1).

Again, further analysis and simulation may be carried out with resulting changes in the specification. We will again assume that none (improbable) are needed, and get on with our development process.

The next step will involve mapping our elaborated functions onto "hardware." We must digress in our next example to produce the specifications of a piece of "hardware."

$STORAGE\text{-}A_{kw}\text{-}PARTITION \equiv (STORAGE\text{-}A_k, CELL_0\text{-}A, .., CELL_k\text{-}A)$

$STORAGE\text{-}A_k \equiv (STORAGEF\text{-}A, \{k\})$

$STORAGEF\text{-}A(k) \equiv [NOMSG(MSG):k, T:P_1^2\ (k, [GRT(ADD,k):XCSA_2(F),$

$T: [EQ(ADD,0):XCSAI_0(OP), .., T: XCSAI_k(OP)]])]$

where $MSG \equiv XSSA_1^1(F)$, $ADD \equiv P_2^2(MSG)$, $OP \equiv P_1^2(MSG)$

$CELL_i\text{-}A \equiv (CELLF_i\text{-}A, N_w)$

$CELLF_i\text{-}A(m) = [EQ(XCSAI_1(F),1):XCSA_2(F), T: P_1^2\ (m, XCSA_2(m))]$

$XCSAI_i$: $\{F\} \rightarrow \{0,1\}$, $\{0,1\} \rightarrow \{F\}$

$XSSA_1$: $\{F\}\ \{0,1\} \times N_{k_7}$ $\qquad$ $XASA_1$: $\{0,1\} \times N_{k_7} \rightarrow \{F\}$

$XCSA_2$: $N_{k_7} \rightarrow \{F\}$, $\{F\} \rightarrow N_{k_7}$

Figure 4.7-2. Definition Set 6: Storage-A_{kw}-Partition.

4.7 A Storage Model

One of the less complex pieces of hardware is a storage unit. Figure 4.7-1 describes one such model. There is a control process STORAGE-A_{kw} and k+1 cell processes. Each cell will hold one value, an integer of maximum value w.

The storage-A_{kw} process receives an access command via $XSSA_1$ to read or write. At most one such command can arrive in a given step. If the address is out of range (there are only k+1 cells) the control unit returns a false-valued message to the requesting process. No synchronization is required for the initial access command. The subsequent transfer of information is synchronized to lock out other access commands until the initial one has been serviced. It is assumed that the accessing process will execute either a read or write function as specified in Figure 4.7-2. If the address is in range, the addressed cell is sent an "operation code" of 0 or 1 for read or write.

Each $cell_i$-A has a state specified by an integer within range. The cell is synchronized and waits for the control process when an access is to be made. Upon receipt of the "operation code," the cell transfers the desired information and resumes its normal cycle of remembering "m."

The storage-A_{kw} is a tuple of processes but is not a system since those processes do not form a complete specification. The interface to the partition is specified by the

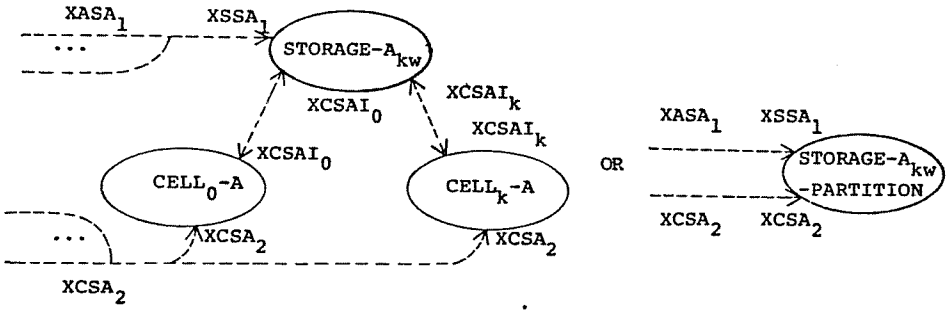


Figure 4.7-1. Storage-A_{kw}-Partition. This is an exchange graph of a model of a STORAGE-A-unit with k+1 words of maximum value w. This figure represents k+2 processes which are one element of a system partition. Any process in the system may read or write by executing the following:

READ-A:$N_k \to \{F\} \cup N_w$ "READ-A(a) = if a > w then false else m"

READ-A(ADD) $\equiv CXSA_2(XASA_1((0,ADD)))$

WRITE-A: $N_k \times N_w \to \{F\}$ "always returns false."

WRITE-A(ADD,m) $\equiv XCSA_2(m,XASA_1((1,ADD))))$.

PATIENT-MONITORING-SYSTEM$_4$ $\equiv$ (CLOCK,MONITOR,DATA-BASE,NURSE-STATION,

NURSE$_1$,..,NURSE$_{k_8}$,PATIENT$_1$,BUFFER$_1$,..,PATIENT$_{k_1}$,BUFFER$_{k_1}$)

MONITOR $\equiv$ (MONITORF,MONITORS) "uses a linear schedule."

MONITORF ((TIME,i),RANGE,PAR) $\equiv$ [GRT(TIME, XACK^1(F)):((TIME,i),RANGE,PAR),

T: MON(XACK^1(F),i,RANGE,((NEXT$_1$,DEL$_1$),..,(SUM(XACK^1(F),DEL$_i$),DEL$_i$),

..,(NEXT$_{k_1}$,DEL$_{k_1}$)), [EQ(i,1): XCM$_1$(F),..,T: XCM$_{k_1}$(F)]]

where PAR $\equiv$ ((NEXT$_1$,DEL$_1$),..,(NEXT$_i$,DEL$_i$),..,(NEXT$_{k_1}$,DEL$_{k_1}$))

MON(TIME,i,RANGE,PAR) $\equiv$ (MIN(NEXT$_1$,..,NEXT$_{k_1}$),

P_1^2 (RANGE, XCB((TIME,i),FACTORS))),

P_1^2 (PAR, [EQ(i,1): Y_1,EQ(i,2):Y_2,..,T:y_k]))

where $Y_i \equiv XCD_i$ (OUT-OF-RANGE(FACTORS,P_1^k (RANGE)))^1

OUT-OF-RANGE((X$_0$,..,X$_{k_2}$),((L$_0$,H$_0$),..,(L$_{k_2}$,H$_{k_2}$))) $\equiv$ (TEST(X$_0$,L$_0$,H$_0$),..,TEST(X$_{k_2}$,L$_{k_2}$,H$_{k_2}$)

TEST(X,L,H) $\equiv$ EOR (GRT(L,X),GRT(X,H)) "tests may be done in parallel."

SUM: $N_{k_3} \times N_{k_3} \to N_{k_3}$ "is modular, integer sum."

MIN: $\prod_{1 \le i \le k_1} N_{k_3} \to$ NEXT-OBS "let x_i = MINIMUM(x_1,..,x_{k_1}) then MIN(X) = (x_i,i)."

EOR: B $\times$ B $\to$ B "exclusive or."

Figure 4.6-1. Definition Set 5: Patient-Monitoring-System$_4$.

$XSSA_1$ and $XCSA_2$ functions. The $XCSAI_i$ functions are local to the partition.

We may now get our process specifications onto somewhat more familiar ground for programmers or hardware designers by implementing some of our patient-monitor-system$_4$ functions in terms of a storage unit as in the next example.

4.8 Process and Storage Partitioning

We can now use our storage specification to contain state information in a conventional way. We now introduce a conventional (ALGOL-like) language shorthand for declaring data structures and variable references. A variable name may be subscripted or unsubscripted (e.g., X, X[i,j]) and the variable names must be declared as part of some data structure that maps them onto some storage in a conventional way. The actual expression represented by a variable name is a function that can be automatically generated (e.g., FACTORS [j] ≡ SUM(FACTORS,j)). Data structures and variables are convenient abstractions of storage process interactions. Note that it is only now that conventional programming language constructs of variable and assignment are meaningful abstractions of the specifications. Their specification and elaboration can thus be deferred until the required contexts and performances have been developed. Now that storage processes have been introduced to our design, we can study questions of data structure and allocation to various storage units. Such allocation introduces additional performance couplings.

Except for monitor, buffer$_i$, and nurse-station, all other processes are left unchanged. The patient, clock, and database interactions with monitor are also unchanged. Nurse$_j$ and nurse-station interactions are the same as before.

The $buffer_i$ processes have now been mapped as a vector in storage, with storage itself serving as the buffer. The nurse-station has changed only by "reading" buffers from storage instead of the $buffer_i$ processes.

The only major changes are in the monitor process that now has a null process state and serves only as a "processor" operating on storage. All value access to former state variables is via read and write functions. There are some arguments, local to monitor, that do not appear in storage. Ultimately these would be mapped to temporary cells in some storage or into registers in a "processor." MONITORF is the same function here as it was in the previous specification, although here it is specified in terms of operations (read, write) on a storage. We must now introduce "type" conversion to store our notices into an integer memory.

Our specifications may now be analyzed and simulated in substantial detail. We could test many performance requirements and will have generated quite detailed performance specifications by now. We could now continue with our design by transforming the monitor process to hardware as a processor, to firmware as a microprogram, or to software as a program placed in storage and interpreted by some processor. All of these alternatives can be easily specified in the same way as our other specifications. The uniformity of the specifications, from requirements to hardware, allows us to use the

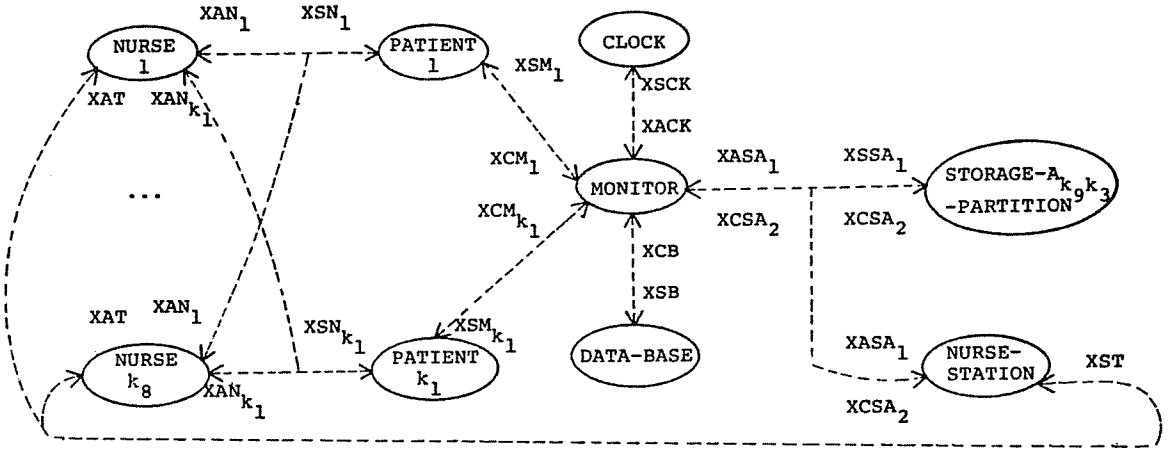


Figure 4.8-1. Patient-Monitor-System$_5$. This is an exchange graph of a system incorporating a storage partition to hold state information of the monitor and nurse-station processes. Its use replaces the previous buffer processes. Counting the storage partition as one "process," this figure represents $k_8 + k_1 + 5$ processes.

PATIENT-PARTITIONING-SYSTEM$_j$ ≡ (CLOCK,MONITOR,DATA-BASE,NURSE-STATION,
STORAGE-A$_{k_3}$-PARTITION,PATIENT$_1$,...,PATIENT$_{k_1}$)

MONITOR ≡ (MONITORF,i)) "a control process with null state."

MONITOR( ) ≡ [GRT(READ-A(TIME,XACK^1(F)));, T:
NULL(RNG(F_1^2 (XACK^1(F),WRITE-A(FAR-NEXT[READ-A(PID)],CON(READ-A
(FAR-DEL[READ-A(PID)]),XACK^1(F)))),
F_1^2 (READ-A(PID),WDTA(READ-A(PID))))]

WDTA(i) ≡ (Y_{i0},Y_{i1},...,Y_{ik_2})
where Y_{i1} ≡ WRITE-A(FACTORS[j], V_{ij})
and Y_{ij} ≡ $P_j^{k_2+1}$ (XCK$_1^1$(F))

MON(t,i) ≡ (WRITE-A(TIME,F_1^2 (SMIN^1( ))),
WRITE-A(PID,F_2^2 (SMIN^1( ))),
WRITE-A(BUFFER[i], OUT-OF-RANGE(i),
XCH(((t,j),(Z_0,..,Z_{k_2}))))
where Z_j ≡ READ-A(FACTORS[j])

OUT-OF-RANGE(i) ≡ CONV((TEST(i,0),...,TEST(i,k_2)))

TEST(i,j) ≡ LOR(GRT(READ-A(RANGEL[i,j]),READ-A(FACTOR[j])),
GRT(READ-A(FACTOR[j]),READ-A(RANGEH[i,j])))
"all tests may be done in parallel."

NURSE-STATION ≡ (NURSE-STATIONF,H_{k_3})

NURSE-STATION(j) ≡ F_1^2 ([EQ(j,k_1):1,T: SUC(j)],
XCT((j,[EQ(j,1):READ-A(BUFFER[1]),..,T:READ-A(BUFFER[k_1])])))

SMIN() ≡ MIN(READ-A(FAR-NEXT[1],..,READ-A(FAR-NEXT[k_1]))

READ(ADD) ≡ XCSA$_2$(F_1^2(r,XASA$_j$((1,ADD))))

WRITE(ADD,m) ≡ XCSA$_2$(XASA$_1$((0,ADD)))

CONV: $\prod_{1\le i\le k_2}$ B → H_{k_3} "converts Boolean tuple into integer."

NULL: Z → { } "maps any set into null set."

k_9 ≡ $5k_1 + 2k_1k_2 + k_2 + 3 \in H_{k_3}$ "is size of STORAGE-A in number of integers."

XASA$_1$: {0,1} × H_{k_9} → {F} XSSA$_1$: {F} → {0,1} × H_{k_9}

XCSA$_2$: H_{k_9} → {F}, {F} → H_{k_9}

"a use of variable name denotes an integer to be interpreted as an address in storage via the usual mappings defined by a declared data structure and should be considered as a shorthand notation for the actual expression to be evaluated. Variable names may be subscripted, i.e. FACTORS[j]. We give an ALGOL-like declaration of the variables used here."

STORAGE-A contains
INTEGER TIME,PID; "INTEGER Y implies 0≤Y<2^{31}."
INTEGER ARRAY RANGEL,RANGEH [1:k_1,0:k_2];
INTEGER ARRAY FACTORS[0:k_2];
INTEGER ARRAY FAR-NEXT,FAR-DEL,BUFFER[1:k_1];

Figure 4.8-2. Definition Set 7: Patient-Monitoring-System$_j$.

same analysis and simulation tools from start to finish. Most importantly, the customer may study the design, in his terms, by simulation at each step of the development process. The designer may also do that and can achieve substantial and early feedback on the consequences of his decisions. The early specification of nonlocal interactions facilitates subsequent system partitioning with confidence that parallel elaborations are still logically independent. Some performance information can be obtained even at early stages.

4.9 Conclusions

We have introduced a specification formalism based on both "exchange" functions and asynchronous processes, and illustrated their use in a sequence of specifications linked by a hypothetical development process. We have shown that the highly mathematical specification language easily extends the domain of discussion to include a large and interesting class of asynchronous processes.

We now have a useful and common language in which to specify, discuss, and solve many interesting system development problems. Such specifications have been designed to make analysis and simulation possible if not always easy. Clearly, such an analysis, even when trivial, can be accurately done only when automated. These specifications must (and can) be checked by such automated tools prior to confidence that they really are complete, consistent, etc. These tools, although made possible by this formalism, have not been implemented as yet. Consequently, errors may exist in the example specifications.

We believe the sequence of examples has shown that formal specifications may be practical even at the earliest stages of development where primitives are very high level indeed.

The sequence of examples has also shown that asynchronous processes (with their freedom from premature details of hardware or software) are generally applicable to specifying systems at any level of abstraction.

This specification formalism is useful to system developers in its own right, without prejudice as to the design methods and theories employed. As a consequence, our specifications make an excellent basis for further study of design methods and theory.

We are currently pursuing such studies on a broad front and many preliminary results have been developed [FD]. Exchange functions have also been used in another approach [DW] to requirements specifications. We invite others to try out both this notation and asynchronous processes. We would be pleased to see your results.

[DW] DEWOLF, J. Barton, and Principato, Robert N. "A Methodology for Requirements Specification and Preliminary Design of Real-Time Systems." Report C-4923, July 1977, The Charles Stark Draper Laboratory, Inc., Cambridge, Mass.

5. Functional Specification of a Microprocessor

5.1 Introduction

This section gives a formal high-level functional specification for a particular configuration of the Motorola M6800 hardware microprocessor. The notation used is that of the specification source language of section 3.4.3.2 with minor modifications as noted in section 5.2. In such a high-level description we functionally designate transitions between well-defined states of the component sub-processors of the microprocessor. This means that many possible hardware designs besides that of the M6800 are capable of realizing our specifications at the physical implementation level. In contrast, a very low-level specification could, for example, represent every logic gate and control signal within the hardware and thus allow for far less flexibility in hardware design choice. Presumably in a top-down design scheme we would arrive at the present level of specification only very late in the complete design process. Further, we would not expect to generate functional specifications in order to match previously existing hardware as we have done here since this constitutes, in effect, a design process in reverse. Our intent in this and other examples is to show how functional specifications can be applied over the entire spectrum of design steps in a top-down design process.

The M6800 microprocessor was chosen as our example for demonstration of the use of functional specifications on the basis of its simplicity of design, especially in that memory and peripherals are addressed uniformly on the same bus by the central processor. Only the most basic information on the M6800 will be reproduced here from [M75] as we elaborate our specifications. Reference to Chapters 1 and 3 of that document will be necessary for a thorough comprehension of the specifications presented throughout this section. The particular configuration of component processors which our specifications define consists of the central processor (or MPU for "microprocessing unit"), a random access memory (RAM), a read only memory (ROM), and a peripheral interface adapter (PIA). We have omitted specifications for a reset switch and a power-down switch (both trivial) and for two peripheral devices which may be attached to the PIA, such as a CRT or a line printer.

Our processes can be represented graphically by an interaction diagram (Figure 5-1) which illustrates all exchange functions used in subsequent specifications. Reset signals and power-down signals are sent to the MPU via the channels RES and NMI (non-maskable interrupt), respectively. The chan-

---

[M75] M6800 Microprocessor Applications Manual, Motorola Semiconductor Products, Inc., 1975.

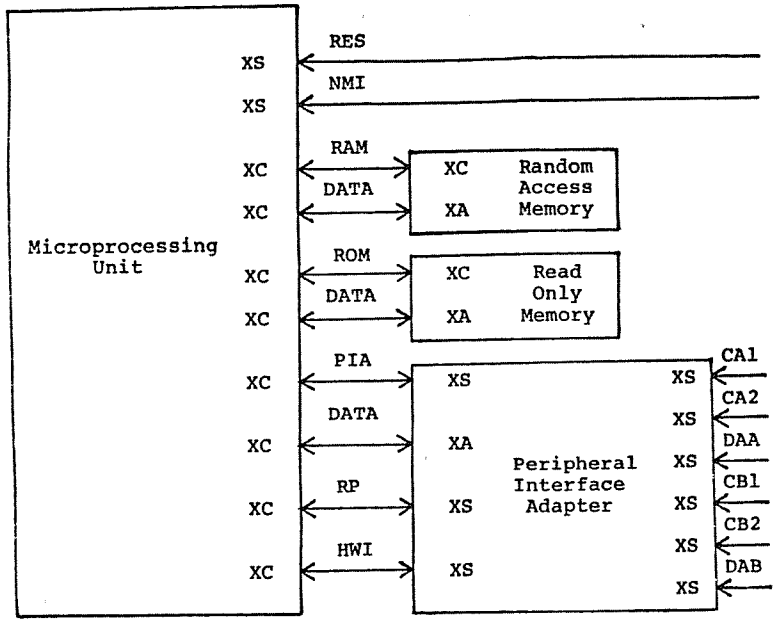


Figure 5-1. Exchange function interactions for a microprocessing system.

nels RAM, ROM, and PIA each send addresses and read/write signals to the processors of the same name, and channel DATA sends data between the MPU and the other three processors. The remaining channels all involve the PIA. Specifically, the MPU resets the PIA via RP and HWI sends interrupt status information from the PIA to the MPU. In addition, CA1, CA2,

CB1, and CB2 are control channels for peripheral devices, whereas DAA and DAB are data channels with CA1, CA2, DAA reserved for one device and CB1, CB2, and DAB reserved for a second device. In the definitions which follow we use numerical subscripts on exchange functions rather than the mnemonic symbols. This is purely a convenience and is not required by the specification source language since named constants (having distinct but unspecified values) are also acceptable. The correspondences are as follows, where numerical values in parentheses are followed by their mnemonic equivalents: (1) RES, (2) RP, (3) NMI, (4) HWI, (5) RAM, (6) ROM, (7) PIA, (8) DATA, (0,1) CA1, (0,2) CA2, (0,3) DAA, (1,1) CB1, (1,2) CB2, and (1,3) DAB.

5.2 Notation and Organization

Before we elaborate each of the four microcomputer processes in turn, thus clarifying the use of the exchange functions shown in Figure 5-1 it is necessary to introduce the notation and basic functions which will be used throughout our specifications. The notation is essentially that of section 3.4.3.2; however, a few differences should be noted. For the sake of brevity in section 5.3 we have permitted the addition and subtraction of subscripts, as in ordinary algebraic notation. This augmentation of the language would surely not complicate in any significant way the analyses of specifications using the feature, and so it is justified by the ef-

fort it saves. Another change is purely notational and involves the iterator construct explained in note (4) of section 3.4.3.2. We will in section 5 simply invert the order of the terminals and substitute "<" and ">" for "(" and ")" respectively, so that, for example, the list a_1,a_2,a_3 is represented via iteration not as $i_1(_3,a_i)$ but as $_1i_3<,a_i>$ instead. Another modification of the source language is the use of the semicolon in place of the comma after pairs of functional expressions in selector functions (see section 3.4.3.2, notes (39-41)). These latter two changes to the source language are trivial and do not add to the complexity of parsing or analyzing specifications. Finally, we have used expressions of the form "replacing α by β" within definitions where attributes (i.e., comments) would normally appear. Each such expression merely indicates the textual substitution of β for α everywhere within the definition. This device saves space and writing effort and again does not add to the complexity of analyzing specifications.

We have not written our microprocessor specification as a sentence in the specification source language of section 3.4.3.2 since we have interleaved lists of formal definitions and informal English language text for each process. This is necessary because of the great length of the formal definition, which is primarily of a pedagogical nature and so is organized accordingly. On the other hand, a formal specification entire-

ly in the source language can easily be constructed from our definitions by placing the four successor function definitions for the MPU, RAM, ROM, and PIA in a specification definition (section 3.4.3.2, note (20)) along with (here non-existent but easily supplied) definitions for a reset switch, power-down switch, and two peripherals. This specification and all remaining definitions need then only to be embedded within a block definition (section 3.4.3.2, note (2)) in order to produce a complete, syntactically correct specification in the source language, as augmented in this section. It should be noted that name scoping rules apply to our existing definitions as if they were in fact included in a single block. That is, definitions in one list may refer to definitions in another list.

The functions introduced in section 5.3 are used repeatedly throughout the rest of section 5, but are of a miscellaneous character and so are included as a unit. They may thus be passed over by the reader and then referenced later where appropriate function names from that group appear in subsequent sections. The function names tend to be mnemonic and fairly self-explanatory. For example, Eor is the exclusive or function, and Ovflow is the arithmetic overflow function. Note that although we have mentioned a purely arithmetic concept and will introduce a function called Add, these are interpretations of values consisting solely of bit vectors. Integers as such do not occur in the present low level of

abstraction; however, as we have mentioned, it is possible to have mapped integers onto bit vectors at some earlier design stage. Because the M6800 is a byte-oriented processor with 16-bit addressing we have represented the states of the processors primarily as vectors of 8-bit strings (8-tuples) and 16-bit strings (16-tuples). Presumably integers or floating point numbers were mapped onto bit strings at an earlier stage of the design and so are not appropriate for the current description of process states.

$1^{n}{}_{17}$<,Let $V_n \equiv 1^{m}{}_{n}$<×{0,1}>>,

$1^{n}{}_{16384}$<,$1^{j}{}_{n}$<,$_{j}k_{n}$<,Let $Pr_{j,k,n}: V_n \to V_{k-j+1}$ where

$Pr_{j,k,n}(1^{\ell}{}_{n}<,x_{\ell}>) \equiv ({}_{j}m_{k}<,x_{m}>)$>>>,

$1^{h}{}_{107}$<,$1^{i}{}_{17}$<,Let $Jn_{h,i}: V_h \times V_i \to V_{h+i}$ where

$Jn_{h,i}((1^{p}{}_{h}<,x_{p}>),(_{h+1}q_{h+i}<,x_{q}>)) \equiv (1^{r}{}_{h+i}<,x_{r}>)$,

$1^{j}{}_{17}$<Let $Jn_{h,i,j}: V_h \times V_i \times V_j \to V_{h+i+j}$ where

$Jn_{h,i,j}(t,u,v) \equiv Jn_{h+i,j}(Jn_{h,i}(t,u),v)$,

$1^{k}{}_{17}$<Let $Jn_{h,i,j,k}: V_h \times V_i \times V_j \times V_k \to V_{h+i+j+k}$ where

$Jn_{h,i,j,k}(t,u,v,w) \equiv Jn_{h+i+j,k}(Jn_{h,i,j}(t,u,v),w)$,

$1^{\ell}{}_{16}$<Let $Jn_{h,i,j,k,\ell}: V_h \times V_i \times V_j \times V_k \times V_\ell \to$

$V_{h+i+j+k+\ell}$ where $Jn_{h,i,j,k,\ell}(t,u,v,w,x) \equiv$

$Jn_{h+i+j+k,\ell}(Jn_{h,i,j,k}(t,u,v,w),x)$,

$1^{m}{}_{16}$<Let $Jn_{h,i,j,k,\ell,m}: V_h \times V_i \times V_j \times V_k \times V_\ell \times V_m \to$

$V_{h+i+j+k+\ell+m}$ where $Jn_{h,i,j,k,\ell,m}(t,u,v,w,x,y) \equiv$

$Jn_{h+i+j+k+\ell,m}(Jn_{h,i,j,k,\ell}(t,u,v,w,x),y)$,

$1^{n}{}_{8}$<Let $Jn_{h,i,j,k,\ell,m,n}: V_h \times V_i \times V_j \times V_k \times V_\ell \times V_m \times V_n \to$

$V_{h+i+j+k+\ell+m+n}$ where $Jn_{h,i,j,k,\ell,m,n}(t,u,v,w,x,y,z) \equiv$

$Jn_{h+i+j+k+\ell+m,n}(Jn_{h,i,j,k,\ell,m}(t,u,v,w,x,y),z)$>>>>>>>,

Let Eq_1: Integer×Integer → Boolean with `$Eq_1(x,y)$ ≡

True if x equals y, otherwise $Eq_1(x,y) \equiv$ False´,

$2^{n}{}_{14}$<, Let $Eq_n: V_n \times V_n \to$ Boolean where $Eq_n(x,y) \equiv$

$[Eq_{n-1}(Pr_{1,n-1,n}(x),Pr_{1,n-1,n}(y))$:

$[Eq_1(Pr_{n,n,n}(x),Pr_{n,n,n}(y)$: True, False]; False],?

Let Equiv: $V_1 \times V_1 \to V_1$ where Equiv(x,y) ≡

[Eq_1(x,y): 1;0],

$1^n{}_{14}$<,Let Neq_n: $V_n \times V_n \to$ Boolean where

Neq_n(x,y) ≡ [Eq_n(x,y): False; True]>,

Let $Comp_1$: $V_1 \to V_1$ where

$Comp_1$(x) ≡ [Eq_1(x,0):1;0],

$2^n{}_{16}$<,Let $Comp_n$: $V_n \to V_n$ where $Comp_n$(x) ≡

$Jn_{n-1,1}$($Comp_{n-1}$($Pr_{1,n-1,n}$(x)),$Comp_1$($Pr_{n,n,n}$(x)))>,

Let And_1: $V_1 \times V_1 \to V_1$ where And_1(x,y) ≡

[Eq_1(x,0): 0; Eq_1(y,0): 0; 1],

$2^n{}_{16}$<,Let And_n: $V_n \times V_n \to V_n$ where And_n(x,y) ≡

$Jn_{n-1,1}$(And_{n-1}($Pr_{1,n-1,n}$(x),$Pr_{1,n-1,n}$(y)),

And_1($Pr_{n,n,n}$(x),$Pr_{n,n,n}$(y)))>,

Let Or_1: $V_1 \times V_1 \to V_1$ where Or_1(x,y) ≡

[Eq_1(x,1):1; Eq_1(y,1): 1; 0],

$2^n{}_{16}$<,Let Or_n: $V_n \times V_n \to V_n$ where Or_n(x,y) ≡

$Jn_{n-1,1}$(Or_{n-1}($Pr_{1,n-1,n}$(x),$Pr_{1,n-1,n}$(y)),

Or_1($Pr_{n,n,n}$(x),$Pr_{n,n,n}$(y)))>,

Let Eor_1: $V_1 \times V_1 \to V_1$ where Eor_1(x,y) ≡ $Comp_1$(Equiv(x,y)),

$2^n{}_{16}$<,Let Eor_n: $V_n \times V_n \to V_n$ where Eor_n(x,y) ≡

$Jn_{n-1,1}$(Eor_{n-1}($Pr_{1,n-1,n}$(x),$Pr_{1,n-1,n}$(y)),

Eor_1($Pr_{n,n,n}$(x),$Pr_{n,n,n}$(y)))>,

Let Add_1: $V_3 \to V_2$ where Add_1(x) ≡

[Eq_3(x,(0,0,0)): (0,0); Eq_3(x,(0,0,1)): (0,1);

Eq_3(x,(0,1,0)): (0,1); Eq_3(x,(0,1,1)): (1,0);

Eq_3(x,(1,0,0)): (0,1); Eq_3(x,(1,0,1)): (1,0);

Eq_3(x,(1,1,0)): (1,0); (1,1)],

$2^n{}_{16}$<,Let Add_n: $V_n \times V_n \times V_1 \to V_{n+1}$ where

Add_n(x,y,k) ≡ $Jn_{n-1,1}$(Add_{n-1}($Pr_{1,n-1,n}$(x),$Pr_{1,n-1,n}$(y),

$Pr_{1,1,2}$(z)),$Pr_{2,2,2}$(z)) replacing "z" by

"Add_1($Pr_{n,n,n}$(x),$Pr_{n,n,n}$(y),k)">,

$1^n{}_{16}$<,Let Ad_n: $V_n \times V_n \times V_1 \to V_n$ where

Ad_n(x,y,k) ≡ $Pr_{2,n+1,n+1}$(Add_n(x,y,k))>,

$1^n{}_{16}$<,Let $Zero_n$: $V_n \to V_1$ where $Zero_n$(x) ≡

[Eq_n(x,($_1i_n$<,0>)): 1; 0]>,

Let Zero: $V_8 \to V_1$ where Zero(x) ≡ $Zero_8$(x),

Let Sign: $V_8 \to V_1$ where Sign(x) ≡ $Pr_{1,1,8}$(x),

Let Carry: $V_9 \to V_1$ where Carry(x) ≡ $Pr_{1,1,9}$(x),

Let Ovflow: $V_8 \times V_8 \times V_8 \to V_1$ where Ovflow(x,y,z) ≡

(And_1(Equiv($Pr_{1,1,8}$(x),$Pr_{1,1,8}$(y)),

Equiv($Pr_{1,1,8}$(x),$Comp_1$($Pr_{1,1,8}$(z)))),

Let Sc_1: $V_{16} \to V_{16}$ where Sc_1(x) ≡ Ad_{16}(x,($_1i_{16}$<,0>),1),

$2^n{}_8$<,Let Sc_n: $V_{16} \to V_{16}$ where Sc_n(x) ≡ $_1i_n$<∘Sc_1(x)>>,

Let Pd_1: $V_{16} \to V_{16}$ where Pd_1(x) ≡ Ad_{16}(x,($_1i_{16}$<,1>),0),

$2^n{}_8$<,Let Pd_n: $V_{16} \to V_{16}$ where Pd_n(x) ≡ $_1i_n$<∘Pd_1(x)>>

5.4 The Central Processor

5.4.1 Overview

The central processor, or MPU, is by far the most complex of the four processors in our microcomputer system. However, its similarity to other conventional central processors makes the task of understanding its functional definition quite straightforward. We have shown already in section 5.1 that the MPU can read or write from memory or peripheral devices and can respond to interrupts via exchange functions. Further elaboration will show how the MPU executes instructions in conjunction with memory and peripherals as well. In particular, the MPU successor function Sumpu specifies how the MPU reacts to interrupt signals (via XS_{RES}, XS_{NMI}, and XS_{HWI}) by branching to instructions stored in high addresses of the ROM. If no interrupt signals are encountered during a system step then the MPU reads a byte from memory and decodes that byte as an operation code. If the operation code falls within one of the types corresponding to six possible operand addressing modes then further action is taken to complete the system step, which corresponds to completion of a machine instruction. Any other operation code results in a state of the MPU not specified in [M75], and so we cannot elaborate further the primitive function Badcode which denotes the successor function in the latter case. In the equations of section 5.4.2 the A and B represent the two 8-bit accumulators; C is the 6-bit

condition register; I is the 16-bit index register; P is the 16-bit program counter; and S is the 16-bit stack pointer. Lower case versions of these letters are used for the same purpose in section 5.4.3. Section 5.4.2 deals with interprocessor interaction and instruction decoding and section 5.4.3 deals exclusively with definitions of individual machine instructions.

5.4.2 Definitions for Interprocessor Interaction and Instruction Decoding

Let Mpu ≡ $V_8 \times V_8 \times V_6 \times V_{16} \times V_{16} \times V_{16}$,

Let Sumpu: Mpu → Mpu where Sumpu(A,B,C,I,P,S) ≡

[$Eq_1(XS_1(0),1)$: Reset((A,B,C,I,P,S),$XC_2(1)$);

$Eq_1(XS_3(0),1)$: Nmint((A,B,C,I,P,S),Stack);

$Eq_1(And_1(XC_4(0),Comp_1(Pr_{2,2,6}(C))),1)$:

Hwint((A,B,C,I,P,S),Stack); Decode(A,B,C,I,P,S)]

replacing "Stack" by "Write(Pd_6(S),(0,0,C)),

Write(Pd_5(S),B),Write(Pd_4(S),A),Write(Pd_3(S),

$Pr_{1,8,16}$(I)),Write(Pd_2(S),$Pr_{9,16,16}$(I)),Write(Pd_1(S),

$Pr_{1,8,16}(Sc_1(P))$),Write(S,$Pr_{9,16,16}(Sc_1(P))$)",

Let Write: $V_{16} \times V_8 \rightarrow V_8$ where Write(x,y) ≡

$XC_8(Pr_{1,8,51}(Jn_{8,17,17,17}(y,XC_5(Jn_{1,16}(1,x)),$

$XC_6(Jn_{1,16}(1,x)),XC_7(Jn_{1,16}(1,x)))))$,

Let XS_1: $V_1 \rightarrow V_1$, Let XS_2: $V_1 \rightarrow V_1$, Let XC_3: $V_1 \rightarrow V_1$,

Let XC_4: $V_1 \rightarrow V_1$, ${}_5n_7$<Let XC_n: $V_{17} \rightarrow V_{17}$>,

Let XC_8: $V_8 \rightarrow V_8$,

Let Reset: Mpu×V_1 → Mpu where

Reset((A,B,C,I,P,S),R) ≡ (A,B,Setint(C),I,

$Jn_{8,8}$(Read(${}_1i_{15}$<,1>,0),Read(${}_1j_{15}$<,1>,1)),S),

Let Setint: $V_6 \rightarrow V_6$ where Setint(C) ≡

$Jn_{1,1,4}(Pr_{1,1,6}(C),1,Pr_{3,6,6}(C))$,

Let Read: $V_{16} \rightarrow V_8$ where Read(x) ≡

$XC_8(Pr_{1,8,51}(Jn_{8,17,17,17}((_1i_8<,0>),XC_5(Jn_{1,16}(0,x)),$

$XC_6(Jn_{1,16}(0,x)),XC_7(Jn_{1,16}(0,x)))))$,

Let Nmint: Mpu×(${}_1i_7$<×V_8>) → Mpu where

Nmint((A,B,C,I,P,S),X) ≡ (A,B,Setint(C),I,

$Jn_{8,8}$(Read(${}_1j_{14}$<,1>,0,0),Read(${}_1k_{14}$<,1>,0,1)),S),

Let Hwint: Mpu×(${}_1i_7$<×V_8>) → Mpu where

Hwint((A,B,C,I,P,S),X) ≡ (A,B,Setint(C),I,

$Jn_{8,8}$(Read(${}_1j_{13}$<,1>,0,0,0),Read(${}_1k_{13}$<,1>,0,0,1)),S),

Let Decode: Mpu → Mpu where Decode(A,B,C,I,P,S) ≡

[Op(P,1,11): Aba(R_1); Op(P,4,15): Clra(R_1);

Op(P,5,15): Clrb(R_1); Op(P,1,1): Cba(R_1);

Op(P,4,3): Coma(R_1); Op(P,5,3): Comb(R_1);

Op(P,4,0): Nega(R_1); Op(P,5,0): Negb(R_1);

Op(P,1,9): Daa(R_1); Op(P,4,10): Deca(R_1);

Op(P,5,10): Decb(R_1); Op(P,4,12): Inca(R_1);

Op(P,5,12): Incb(R_1); Op(P,3,6): Psha(R_1);

Op(P,3,7): Pshb(R_1); Op(P,3,2): Pula(R_1);

Op(P,3,3): Pulb(R_1); Op(P,4,9): Rola(R_1);

Op(P,5,9): Rolb(R_1); Op(P,4,6): Rora(R_1);

Op(P,5,6): Rorb(R_1); Op(P,4,8): Asla(R_1);

```
Op(P,5,8): Aslb(R1); Op(P,4,7): Asra(R1);
Op(P,5,7): Asrb(R1); Op(P,4,4): Lsra(R1);
Op(P,5,4): Lsrb(R1); Op(P,1,0): Sba(R1);
Op(P,1,6): Tab(R1); Op(P,1,7): Tba(R1);
Op(P,4,13): Tsta(R1); Op(P,5,13): Tstb(R1);
Op(P,0,9): Dex(R1); Op(P,3,4): Des(R1);
Op(P,0,8): Inx(R1); Op(P,3,1): Ins(R1);
Op(P,3,5): Txs(R1); Op(P,3,0): Tsx(R1);
Op(P,0,1): Nop(R1); Op(P,3,11): Rti(R1);
Op(P,3,9): Rts(R1); Op(P,3,15): Swi(R1);
Op(P,3,14): Wai(R1); Op(P,0,12): Clc(R1);
Op(P,0,14): Cli(R1); Op(P,0,10): Clv(R1);
Op(P,0,13): Sec(R1); Op(P,0,15): Sei(R1);
Op(P,0,11): Ser(R1); Op(P,0,6): Tap(R1);
Op(P,0,7): Tpa(R1); Op(P,2,0): Bra(R2);
Op(P,2,4): Bcc(R2); Op(P,2,5): Bcs(R2);
Op(P,2,7): Beq(R2); Op(P,2,12): Bge(R2);
Op(P,2,14): Bgt(R2); Op(P,2,2): Bhi(R2);
Op(P,2,15): Ble(R2); Op(P,2,3): Bls(R2);
Op(P,2,13): Blt(R2); Op(P,2,11): Bmi(R2);
Op(P,2,6): Bne(R2); Op(P,2,8): Brc(R2);
Op(P,2,9): Bvs(R2); Op(P,2,10): Bpl(R2);
Op(P,8,13): Brs(R2); Op(P,9,11): Adda(R3);
Op(P,13,11): Addb(R3); Op(P,9,9): Adca(R3);
Op(P,13,9): Adcb(R3); Op(P,9,4): Anda(R3);
```



```
Op(P,13,4): Andb(R3); Op(P,9,5): Bita(R3);
Op(P,13,5): Bitb(R3); Op(P,9,1): Cmpa(R3);
Op(P,31,1): Cmpb(R3); Op(P,9,8): Eora(R3);
Op(P,13,8): Eorb(R3); Op(P,9,6): Ldaa(R3);
Op(P,13,6): Ldab(R3); Op(P,9,10): Oraa(R3);
Op(P,13,10): Orab(R3); Op(P,9,7): Staa(R4);
Op(P,13,7): Stab(R4); Op(P,9,0): Suba(R3);
Op(P,13,0): Subb(R3); Op(P,9,2): Sbca(R3);
Op(P,13,2): Sbcb(R3); Op(P,9,12): Cpx(R5);
Op(P,13,14): Ldx(R5); Op(P,9,14): Lds(R5);
Op(P,13,15): Stx(R4); Op(P,9,15): Sts(R4);
Op(P,11,11): Adda(R6); Op(P,15,11): Addb(R6);
Op(P,11,9): Adca(R6); Op(P,15,9): Adcb(R6);
Op(P,11,4): Anda(R6); Op(P,15,4): Andb(R6);
Op(P,11,5): Bita(R6); Op(P,15,5): Bitb(R6);
Op(P,7,15): Clr(R7); Op(P,11,1): Cmpa(R6);
Op(P,15,1): Cmpb(R6); Op(P,7,3): Com(R8);
Op(P,7,0): Neg(R8); Op(P,7,10): Dec(R8);
Op(P,11,8): Eora(R6); Op(P,15,8): Eorb(R6);
Op(P,7,12): Inc(R8); Op(P,11,6): Ldaa(R6);
Op(P,15,6): Ldab(R6); Op(P,11,10): Oraa(R6);
Op(P,15,10): Orab(R6); Op(P,7,9): Rol(R8);
Op(P,7,6): Ror(R8); Op(P,7,8): Asl(R8);
Op(P,7,7): Asr(R8); Op(P,7,4): Lsr(R8);
Op(P,11,7): Staa(R6); Op(P,15,7): Stab(R6);
```

Op(P,11,0): Suba(R_6); Op(P,15,0): Subb(R_6);

Op(P,11,2): Sbca(R_6); Op(P,15,2): Sbcb(R_6);

Op(P,7,13): Tst(R_6); Op(P,11,12): Cpx(R_8);

Op(P,15,14): Ldx(R_8); Op(P,11,14): Lds(R_8);

Op(P,15,15): Stx(R_7); Op(P,11,15): Sts(R_7);

Op(P,7,14): Jmp(R_7); Op(P,11,13): Jsr(R_7);

Op(P,10,11): Adda(R_9); Op(P,14,11): Addb(R_9);

Op(P,10,9): Adca(R_9); Op(P,14,9): Adcb(R_9);

Op(P,10,4): Anda(R_9); Op(P,14,4): Andb(R_9);

Op(P,10,5): Bita(R_9); Op(P,14,5): Bitb(R_9);

Op(P,6,15): Clr(R_{10}); Op(P,10,1): Cmpa(R_9);

Op(P,14,1): Cmpb(R_9); Op(P,6,3): Com(R_{11});

Op(P,6,0): Neg(R_{11}); Op(P,6,10): Dec(R_{11});

Op(P,10,8): Eora(R_9); Op(P,14,8): Eorb(R_9);

Op(P,6,12): Inc(R_{11}); Op(P,10,6): Ldaa(R_9);

Op(P,14,6): Ldab(R_9); Op(P,10,10): Oraa(R_9);

Op(P,14,10): Orab(R_9); Op(P,6,9): Rol(R_{11});

Op(P,6,6): Ror(R_{11}); Op(P,6,8): Asl(R_{11});

Op(P,6,7): Asr(R_{11}); Op(P,6,4): Lsr(R_{11});

Op(P,10,7): Staa(R_9); Op(P,14,7): Stab(R_9);

Op(P,10,0): Suba(R_9); Op(P,14,0): Subb(R_9);

Op(P,10,2): Sbca(R_9); Op(P,14,2): Sbcb(R_9);

Op(P,6,13): Tst(R_9); Op(P,10,12): Cpx(R_{11});

Op(P,14,14): Ldx(R_{11}); Op(P,10,14): Lds(R_{11});

Op(P,14,15): Stx(R_{10}); Op(P,10,15): Sts(R_{10});

Op(P,6,14): Jmp(R_{10}); Op(P,10,13): Jsr(R_{10});

Op(P,8,11): Adda(R_{12}); Op(P,12,11): Addb(R_{12});

Op(P,8,9): Adca(R_{12}); Op(P,12,9): Adcb(R_{12});

Op(P,8,4): Anda(R_{12}); Op(P,12,4): Andb(R_{12});

Op(P,8,5): Bita(R_{12}); Op(P,12,15): Bitb(R_{12});

Op(P,8,1): Cmpa(R_{12}); Op(P,12,1): Cmpb(R_{12});

Op(P,8,8): Eora(R_{12}); Op(P,12,8): Eorb(R_{12});

Op(P,8,6): Ldaa(R_{12}); Op(P,12,6): Ldab(R_{12});

Op(P,8,10): Oraa(R_{12}); Op(P,12,10): Orab(R_{12});

Op(P,8,0): Suba(R_{12}); Op(P,12,0): Subb(R_{12});

Op(P,8,2): Sbca(R_{12}); Op(P,12,12): Sbcb(R_{12});

Op(P,8,12): Cpx(R_{13}); Op(P,12,14): Ldx(R_{13});

Op(P,8,14): Lds(R_{13}); Badcode(R_1)]

replacing "R_1" by "A,B,C,I,Sc_1(P),S", replacing "R_2" by "Reladr(A,B,C,I,P,S,Read(Sc_1(P)))",

replacing "R_3" by "Fetchd(R_4)", replacing "R_5" by "Fetchmd(R_4)", replacing "R_4" by "A,B,C,I,Sc_2(P),S, $Jn_{8,8}(({}_1i_8{<},0{>}),Read(Sc_1(P)))$", replacing "$R_6$" by "Fetchd($R_7$)", replacing "$R_8$" by "Fetchmd($R_7$)",

replacing "R_7" by "A,B,C,I,Sc_3(P),S, $Jn_{8,8}(Read(Sc_1(P)),Read(Sc_2(P)))$", replacing "$R_9$" by "Fetchd($R_{10}$)", replacing "$R_{11}$" by "Fetchmd($R_{10}$)",

replacing "R_{10}" by "A,B,C,I,Sc_2(P),S, $Ad_{16}(I,Jn_{8,8}(({}_1i_8{<},0{>}),Read(Sc_1(P))),0)$", replacing "$R_{12}$" by "A,B,C,I,$Sc_2$(P),S,Read($Sc_1$(P))", replacing "$R_{13}$" by "A,B,C,I,$Sc_3$(P),S,$Sc_1$(P),Read($Sc_1$(P))",

```
Let Reladr: Mpud → Mpum where Reladr(A,B,C,I,P,S,D) ≡
   (A,B,C,I,Sc2(P),S,[Eq1(Pr1,1,8(D),1):
      Ad16(Sc2(P),Comp16(Magn),1); Ad16(Sc2(P),Magn,0)]
      replacing "Magn" by "Jn9,7((1i9<,0>),Pr2,8,8(D))",

Let Fetchd: Mpum → Mpud where Fetchd(A,B,C,I,P,S,M) ≡
   (A,B,C,I,P,S,Read(M)),

Let Fetchmd: Mpum → Mpumd where Fetchmd(A,B,C,I,P,S,M) ≡
   (A,B,C,I,P,S,M,Read(M)),

Let Hex ≡ {0i15<,i>},

Let Op: V16×Hex×Hex → Boolean where
   Op(p,x,y) ≡ Eq8(p,Jn4,4(g(x),g(y))),

Let g: Hex → V4 where g(x) ≡
   [1i15<;Eq(x,i): Pr13,16,16(Sci(1j16<,0>))>; (0,0,0,0)],

Let Badcode: Mpu → Mpu with `machine behavior
   unspecified by manufacturer for illegal operation codes´,

Let Mpum ≡ V8×V8×V6×V16×V16×V16×V16,

Let Mpud ≡ V8×V8×V6×V16×V16×V16×V8,

Let Mpumd ≡ V8×V8×V6×V16×V16×V16×V16×V8
```


5.4.3 Definitions for Machine Instructions

```
Let Adda: Mpud → Mpu  where  Adda(a,b,c,i,p,s,d) ≡
         (Ad8(a,d,0),b,Addcond(a,d,0,c),i,p,s),

Let Addcond:  V8×V8×V1×V6 → V6  where  Addcond(x,y,k,c) ≡
         (Pr1,1,5(Add4(Pr5,8,8(x),Pr5,8,8(y),k)),Pr2,2,6(c),
           Sign(z),Zero(z),Ovflow(x,y,z),Carry(Add8(x,y,k)))
      replacing "z" by "Ad8(x,y,k)" ,

Let Addb: Mpud → Mpu  where  Addb(a,b,c,i,p,s,d) ≡
         (a,Ad8(b,d,0),Addcond(b,d,0,c),i,p,s) ,

Let Aba: Mpu → Mpu  where  Aba(a,b,c,i,p,s) ≡
         (Ad8(a,b,0),b,Addcond(a,b,0,c),i,p,s) ,

Let Adca: Mpud → Mpu  where  Adca(a,b,c,i,p,s,d) ≡
         (Ad8(a,d,Pr6,6,6(c)),b,Addcond(a,d,Pr6,6,6(c),c),
          i,p,s) ,

Let Adcb: Mpud → Mpu  where  Adcb(a,b,c,i,p,s,d) ≡
         (a,Ad8(b,d,Pr6,6,6(c)),Addcond(b,d,Pr6,6,6(c),c),
          i,p,s) ,

Let Anda: Mpud → Mpu  where  Anda(a,b,c,i,p,s,d) ≡
         (And8(a,d),b,Logcond(And8(a,d),c),i,p,s) ,

Let Logcond: V8×V6 → V6  where  Logcond(x,c) ≡
         (Pr1,2,6(c),Sign(x),Zero(x),0,Pr6,6,6(c)) ,.
```

Let Andb: Mpud → Mpu where Andb(a,b,c,i,p,s,d) ≡

(a,And_8(b,d),Logcond(And_8(b,d),c),i,p,s) ,

Let Bita: Mpud → Mpu where Bita(a,b,c,i,p,s,d) ≡

(a,b,Logcond(And_8(a,d),c)i,p,s) ,

Let Bitb: Mpud → Mpu where Bitb(a,b,c,i,p,s,d) ≡

(a,b,Logcond(And_8(b,d),c),i,p,s) ,

Let Clr: Mpum → Mpu where Clr(a,b,c,i,p,s,m) ≡

Wr(a,b,$Jn_{2,1,1,1,1}$($Pr_{1,2,6}$(c),a,1,0,0),i,p,s,m,

($_1i_8$<,0>)),

Let Wr: Mpumd → Mpu where Wr(a,b,c,i,p,s,m,d) ≡

$Pr_{1,70,78}$($Jn_{8,8,6,16,16,16,8}$(a,b,c,i,p,s,

Write(m(d)))) ,

Let Clra: Mpu → Mpu where Clra(a,b,c,i,p,s) ≡

(($_1i_8$<,0>,b,$Jn_{2,1,1,1,1}$($Pr_{1,2,6}$(c),0,1,0,0),i,p,s),

Let Clrb: Mpu → Mpu where Clrb(a,b,c,i,p,s) ≡

(a,($_1i_8$<,0>),$Jn_{2,1,1,1,1}$($Pr_{1,2,6}$(c),0,1,0,0),i,p,s),

Let Cmpa: Mpud → Mpu where Cmpa(a,b,c,i,p,s,d) ≡

(a,b,Subcond(a,d,1,c),i,p,s) ,

Let Subcond: $V_8 \times V_8 \times V_1 \times V_6 \rightarrow V_6$ where Subcond(x,y,k,c) ≡

$Jn_{2,1,1,1,1}$($Pr_{1,2,6}$(c),Sign(z),Zero(z),

Ovflow(x,$Comp_8$(y),z),Carry(Add$_8$(x,$Comp_8$(y),k)))

replacing "z" by "Ad_8(x,$Comp_8$(y),k)" ,

Let Cmpb: Mpud → Mpu where Cmpb(a,b,c,i,p,s,d) ≡

(a,b,Subcond(b,d,1,c),i,p,s) ,

Let Cba: Mpu → Mpu where Cba(a,b,c,i,p,s) ≡

(a,b,Subcond(a,b,1,c),i,p,s) ,

Let Com: Mpumd → Mpu where Com(a,b,c,i,p,s,m,d) ≡

Wr(a,b,Compcond($Comp_8$(d),c),i,p,s,m,$Comp_8$(d)) ,

Let Compcond: $V_8 \times V_6 \rightarrow V_6$ where Compcond(x,c) ≡

$Jn_{2,1,1,1,1}$($Pr_{1,2,6}$(c),Sign(x),Zero(x),0,1) ,

Let Coma: Mpu → Mpu where Coma(a,b,c,i,p,s) ≡

($Comp_8$(a),b,Compcond($Comp_8$(a),c),i,p,s) ,

Let Comb: Mpu → Mpu where Comb(a,b,c,i,p,s) ≡

(a,$Comp_8$(b),Compcond($Comp_8$(b),c),i,p,s) ,

Let Neg: Mpumd → Mpu where Neg(a,b,c,i,p,s,m,d) ≡

Wr(a,b,Subcond(($_1i_8$<,0>),$Comp_8$(d),1,c),i,p,s,m,

Ad_8(($_1j_8$<,0>),$Comp_8$(d),1)) ,

Let Nega: Mpu → Mpu where Nega(a,b,c,i,p,s) ≡

(Ad_8(($_1i_8$<,0>),$Comp_8$(a),1),b,Subcond(($_1j_8$<,0>),

$Comp_8$(a),1,c),i,p,s) ,

Let Negb: Mpu → Mpu where Negb(a,b,c,i,p,s) ≡

(a,Ad_8(($_1i_8$<,0>),$Comp_8$(b),1),Subcond(($_1j_8$<,0>),

$Comp_8$(b),1,c),i,p,s) ,

Let Daa: Mpu → Mpu where Daa(a,b,c,i,p,s) ≡

$(z,b,Jn_{2,1,1,1,1}(Pr_{1,2,6}(c),Sign(z),Zero(z),$
$Ovflow(a,t,z),Carry(Ad_8(a,t,0))),i,p,s)$

replacing "z" by "$Add_8(a,t,0)$", replacing "t" by
"$Jn_{4,4}[Eq_1(Or_1(And_1(Adj(v),Eq_4(u,(1,0,0,1))),$
$Or_1(Adj(u),Pr_{6,6,6}(c))):(0,1,1,0)\ ;\ (0,0,0,0)]$,
$[Eq_1(Or_1(Adj(v),Pr_{1,1,6}(c)):(0,1,1,0)\ ;\ (0,0,0,0)])$",
replacing "u" by "$Pr_{1,4,8}(a)$", replacing "v" by
"$Pr_{5,8,8}(a)$" ,

Let Adj: $V_4 \to V_1$ where Adj(x) ≡

$(And_1(Pr_{1,1,4}(x),Or_1(Pr_{2,2,4}(x),Pr_{3,3,4}(x)))$,

Let Dec: Mpumd → Mpu where Dec(a,b,c,i,p,s,m,d) ≡

$Wr(a,b,Inccond(d,({}_1i_8<,1>),0,c),i,p,s,m,$
$Ad_8(d,({}_1j_8<,1>),0))$,

Let Inccond: $V_8 \times V_8 \times V_1 \times V_6 \to V_6$ where Inccond(x,y,k,c) ≡

$Jn_{2,1,1,1,1}(Pr_{1,2,6}(c),Sign(z),Zero(z),$
$Ovflow(x,y,z),Pr_{6,6,6}(c))$

replacing "z" by "$Ad_8(x,y,0)$" ,

Let Deca: Mpu → Mpu where Deca(a,b,c,i,p,s) ≡

$(Ad_8(a,({}_1i_8<,1>),0),b,Inccond(a,({}_1j_8<,1>),0,c),$
$i,p,s)$,

Let Decb: Mpu → Mpu where Decb(a,b,c,i,p,s) ≡

$(a,Ad_8(b,({}_1i_8<,1>),0),Inccond(b,({}_1j_8<,1>),0,c),$
$i,p,s)$,

Let Eora: Mpud → Mpu where Eora(a,b,c,i,p,s,d) ≡

$(Eor_8(a,d),b,Logcond(Eor_8(a,d),c),i,p,s)$,

Let Eorb: Mpud → Mpu where Eorb(a,b,c,i,p,s,d) ≡

$(a,Eor_8(b,d),Logcond(Eor_8(b,d),c),i,p,s)$,

Let Inc: Mpumd → Mpu where Inc(a,b,c,i,p,s,m,d) ≡

$Wr(a,b,Inccond(d,({}_1i_8<,0>),1,c),i,p,s,m,$
$Ad(d,({}_1j_8<,0>),1))$,

Let Inca: Mpu → Mpu where Inca(a,b,c,i,p,s) ≡

$(Ad_8(a,({}_1i_8<,0>),1),b,Inccond(a,({}_1j_8<,0>),1,c),$
$i,p,s)$,

Let Incb: Mpu → Mpu where Incb(a,b,c,i,p,s) ≡

$(a,Ad_8(b,({}_1i_8<,0>),1),Inccond(b,({}_1j_8<,0>),1,c),$
$i,p,s)$,

Let Ldaa: Mpud → Mpu where Ldaa(a,b,c,i,p,s,d) ≡

$(d,b,Logcond(d,c),i,p,s)$,

Let Ldab: Mpud → Mpu where Ldab(a,b,c,i,p,s,d) ≡

$(a,d,Logcond(d,c),i,p,s)$,

Let Oraa: Mpud → Mpu where Oraa(a,b,c,i,p,s,d) ≡
$(Or_8(a,d),b,Logcond(Or_8(a,d),c),i,p,s)$,

Let Orab: Mpud → Mpu where Orab(a,b,c,i,p,s,d) ≡
$(a,Or_8(b,d),Logcond(Or_8(b,d),c),i,p,s)$,

Let Psha: Mpu → Mpu where Psha(a,b,c,i,p,s) ≡
$Wr(a,b,c,i,p,Pd_1(s),s,a)$,

Let Pshb: Mpu → Mpu where Pshb(a,b,c,i,p,s) ≡
$Wr(a,b,c,i,p,Pd_1(s),s,b)$,

Let Pula: Mpu → Mpu where Pula(a,b,c,i,p,s) ≡
$(Read(Sc_1(s)),b,c,i,p,Sc_1(s))$,

Let Pulb: Mpu → Mpu where Pulb(a,b,c,i,p,s) ≡
$(a,Read(Sc_1(s)),c,i,p,Sc_1(s))$,

Let Rol: Mpumd → Mpu where Rol(a,b,c,i,p,s,m,d) ≡
$Wr(a,b,Shifcond(Rota(d,c),c),i,p,s,m,$
$Pr_{2,9,9}(Rotal(d,c)))$,

Let Rotal: $V_8 \times V_6 \to V_9$ where Rotal(x,c) ≡
$Jn_{8,1}(x,Pr_{6,6,6}(c))$,

Let Shifcond: $V_9 \times V_6 \to V_6$ where Shifcond(x,c) ≡
$Jn_{2,1,1,1,1}(Pr_{1,2,6}(c),Sign(y),Zero(y),$
$Eor_1(Sign(y),Carry(x)),Carry(x))$
replacing "y" by "$Pr_{2,9,9}(x)$" ,

Let Rola: Mpu → Mpu where Rola(a,b,c,i,p,s) ≡
$(Pr_{2,9,9}(Rotal(a,c)),b,Shifcond(Rotal(a,c),c),i,p,s)$,

Let Rolb: Mpu → Mpu where Rolb(a,b,c,i,p,s) ≡
$(a,Pr_{2,9,9}(Rotal(b,c)),Shifcond(Rotal(b,c),c),i,p,s)$,

Let Ror: Mpumd → Mpu where Ror(a,b,c,i,p,s,m,d) ≡
$Wr(a,b,Shifcond(Rotar(d,c),c),i,p,s,m,$
$Pr_{2,9,9}(Rotar(d,c)))$,

Let Rotar: $V_8 \times V_6 \to V_9$ where Rotar(x,c) ≡
$Jn_{1,1,7}(Pr_{8,8,8}(x),Pr_{6,6,6}(c),Pr_{1,7,8}(x))$,

Let Rora: Mpu → Mpu where Rora(a,b,c,i,p,s) ≡
$(Pr_{2,9,9}(Rotar(a,c)),b,Shifcond(Rotar(a,c),c),i,p,s)$,

Let Rorb: Mpu → Mpu where Rorb(a,b,c,i,p,s) ≡
$(a,Pr_{2,9,9}(Rotar(b,c)),Shifcond(Rotar(b,c),c),i,p,s)$,

Let Asl: Mpumd → Mpu where Asl(a,b,c,i,p,s,m,d) ≡
$Wr(a,b,Shifcond(Shif(d),c),i,p,s,m,$
$Pr_{2,9,9}(Shifl(d)))$,

Let Shifl: $V_8 \to V_9$ where Shifl(x) ≡ $Jn_{8,1}(x,0)$.

Let Asla: Mpu → Mpu where Asla(a,b,c,i,p,s) ≡
$(Pr_{2,9,9}(Shifl(a)),b,Shifcond(Shifl(a),c),i,p,s)$,

Let Aslb: Mpu → Mpu where Aslb(a,b,c,i,p,s) ≡

$(a,Pr_{2,9,9}(Shifl(b)),Shifcond(Shifl(b),c),i,p,s)$,

Let Asr: Mpumd → Mpu where Asr(a,b,c,i,p,s,m,d) ≡

Wr(a,b,Shifcond(Shifra(d),c),i,p,s,m,

$Pr_{2,9,9}(Shifra(d))$) ,

Let Shifra: $V_8 \rightarrow V_9$ where Shifra(x) ≡

$Jn_{1,1,7}(Pr_{8,8,8}(x),Pr_{1,1,8}(x),Pr_{1,7,8}(x))$,

Let Asra: Mpu → Mpu where Asra(a,b,c,i,p,s) ≡

$(Pr_{2,9,9}(Shifra(a)),b,Shifcond(Shifra(a),c),i,p,s)$,

Let Asrb: Mpu → Mpu where Asrb(a,b,c,i,p,s) ≡

$(a,Pr_{2,9,9}(Shifra(b)),Shifcond(Shifra(b),c),i,p,s)$,

Let Lsr: Mpumd → Mpu where Lsr(a,b,c,i,p,s,m,d) ≡

Wr(a,b,Shifcond(Shifrl(d),c),i,p,s,m,

$Pr_{2,9,9}(Shifrl(d))$) ,

Let Shifrl: $V_8 \rightarrow V_9$ where Shifrl(x) ≡

$Jn_{1,1,7}(Pr_{8,8,8}(x),a,Pr_{1,7,8}(x))$,

Let Lsra: Mpu → Mpu where Lsra(a,b,c,i,p,s) ≡

$(Pr_{2,9,9}(Shifrl(a)),b,Shifcond(Shifrl(a),c),i,p,s)$,

Let Lsrb: Mpu → Mpu where Lsrb(a,b,c,i,p,s) ≡

$(a,Pr_{2,9,9}(Shifrl(b)),Shifcond(Shifrl(b),c),i,p,s)$,

Let Staa: Mpum → Mpu where Staa(a,b,c,i,p,s,m) ≡

Wr(a,b,Logcond(a,c),i,p,s,m,a) ,

Let Stab: Mpum → Mpu where Stab(a,b,c,i,p,s,m) ≡

Wr(a,b,Logcond(b,c),i,p,s,m,b) ,

Let Suba: Mpud → Mpu where Suba(a,b,c,i,p,s,d) ≡

$(Ad_8(a,Comp_8(d),1),b,Subcond(a,d,1,c),i,p,s)$,

Let Subb: Mpud → Mpu where Subb(a,b,c,i,p,s,d) ≡

$(a,Ad_8(b,Comp_8(d),1),Subcond(b,d,1,c),i,p,s)$,

Let Sba: Mpu → Mpu where Sba(a,b,c,i,p,s) ≡

$(Ad_8(a,Comp_8(b),1),b,Subcond(a,b,1,c),i,p,s)$,

Let Sbca: Mpud → Mpu where Sbca(a,b,c,i,p,s,d) ≡

$(Ad_8(a,Comp_8(d),k),b,Subcond(a,d,k,c),i,p,s)$

replacing "k" by "$Comp_1(Pr_{6,6,6}(c))$" ,

Let Sbcb: Mpud → Mpu where Sbcb(a,b,c,i,p,s,d) ≡

$(a,Ad_8(b,Comp_8(d),k),Subcond(b,d,k,c),i,p,s)$

replacing "k" by "$Comp_1(Pr_{6,6,6}(c))$" ,

Let Tab: Mpu → Mpu where Tab(a,b,c,i,p,s) ≡

(a,a,Logcond(a,c),i,p,s) ,

Let Tba: Mpu → Mpu where Tba(a,b,c,i,p,s) ≡

(b,b,Logcond(b,c),i,p,s) ,

Let Tst: Mpud → Mpu where Tst(a,b,c,i,p,s,d) ≡

(a,b,Testcond(d,c),i,p,s) ,

Let Testcond: $V_8 \times V_6 \rightarrow V_6$ where Testcond(x,c) ≡

$Jn_{2,1,1,1,1}(Pr_{1,2,6}(c),Sign(x),Zero(x),0,0)$,

Let Tsta: Mpu → Mpu where Tsta(a,b,c,i,p,s) ≡

(a,b,Testcond(a,c),i,p,s) ,

Let Tstb: Mpu → Mpu where Tstb(a,b,c,i,p,s) ≡

(a,b,Testcond(b,c),i,p,s) ,

Let Cpx: Mpumd → Mpu where Cpx(a,b,c,i,p,s,m,d) ≡

Cpxr(a,b,c,i,p,s,d,Read(Sc_1(m))) ,

Let Mpudd ≡ $V_8 \times V_8 \times V_6 \times V_{16} \times V_{16} \times V_{16} \times V_8 \times V_8$,

Let Cpxr: Mpudd → Mpu where Cpxr(a,b,c,i,p,s,d,e) ≡

(a,b,$Jn_{2,1,1,1,1}(Pr_{1,2,6}(c),Pr_{1,1,16}(z),Zero_{16}(z)$,

Ovflow($P_{9,16,16}$(i),e,$Pr_{9,16,16}(z)$),$Pr_{6,6,6}(c)$),

i,p,s)

replacing "z" by "$Ad_{16}(i,Comp_{16}(Jn_{8,8}(d,e)),1)$" ,

Let Dex: Mpu → Mpu where Dex(a,b,c,i,p,s) ≡

(a,b,$Jn_{3,1,2}(Pr_{1,3,6}(C),Zero_{16}(Pd_1(i)),Pr_{5,6,6}(c))$,

Pd_1(i),p,s) ,

Let Des: Mpu → Mpu where Des(a,b,c,i,p,s) ≡

(a,b,c,i,p,Pd_1(s)) ,

Let Inx: Mpu → Mpu where Inx(a,b,c,i,p,s) ≡

(a,b,$Jn_{3,1,2}(Pr_{1,3,6}(c),Zero_{16}(Xc_1(i)),Pr_{5,6,6}(c))$,

Sc_1(i),p,s) ,

Let Ins: Mpu → Mpu where Ins(a,b,c,i,p,s) ≡

(a,b,c,i,p,Sc_1(s)) ,

Let Ldx: Mpumd → Mpu where Ldx(a,b,c,i,p,s,m,d) ≡

Ldxr(a,b,c,i,p,s,d,Read(Sc_1(m))) ,

Let Ldxr: Mpudd → Mpu where Ldxr(a,b,c,i,p,s,d,e) ≡

(a,b,Doubcond($Jn_{8,8}$(d,e)),$Jn_{8,8}$(d,e),p,s) ,

Let Doubcond: $V_{16} \rightarrow V_6$ where Doubcond(x) ≡

$Jn_{2,1,1,1,1}(Pr_{1,2,6}(c),Sign(Pr_{1,8,16}(x)),Zero_{16}(x)$,

0,$Pr_{6,6,6}(c)$) ,

Let Lds: Mpumd → Mpu where Lds(a,b,c,i,p,s,m,d) ≡

Ldsr(a,b,c,i,p,s,d,Read(Sc_1(m))) ,

Let Ldsr: Mpudd → Mpu where Ldsr(a,b,c,i,p,s,d,e) ≡

(a,b,Doubcond($Jn_{8,8}$(d,e)),i,p,$Jn_{8,8}$(d,e)) ,

Let Stx: Mpum → Mpu where Stx(a,b,c,i,p,s,m) ≡

Wr(Wr(a,b,Doubcond(i),i,p,s,m,$Pr_{1,8,16}$(i)),Sc_1(m),

$Pr_{9,16,16}$(i)) ,

```
Let Sts: Mpum → Mpu  where  Sts(a,b,c,i,p,s,m) ≡
        Wr(Wr(a,b,Doubcond(s),i,p,s,m,Pr_{1,8,16}(s)),Sc_1(m),
         Pr_{9,16,16}(s)) ,

Let Txs: Mpu → Mpu  where  Txs(a,b,c,i,p,s) ≡
        (a,b,c,i,p,Pd_1(i)) ,

Let Tsx: Mpu → Mpu  where  Tsx(a,b,c,i,p,s) ≡
        (a,b,c,Sc_1(s),p,s) ,

Let Bra: Mpum → Mpu  where  Bra(a,b,c,i,p,s,m) ≡
        (a,b,c,i,m,s) ,

Let Bcc: Mpum → Mpu  where  Bcc(a,b,c,i,p,s,m) ≡
        (a,b,c,i,[Eq_1(Pr_{6,6,6}(c),0) : m; p],s) ,

Let Bcs: Mpum → Mpu  where  Bcs(a,b,c,i,p,s,m) ≡
        (a,b,c,i,[Eq_1(Pr_{6,6,6}(c),1) : m; p],s) ,

Let Beq: Mpum → Mpu  where  Beq(a,b,c,i,p,s,m) ≡
        (a,b,c,i,[Eq_1(Pr_{4,4,6}(c),1) : m; p],s) ,

Let Bge: Mpum → Mpu  where  Bge(a,b,c,i,p,s,m) ≡
        (a,b,c,i,[Eq_1(Eor_1(Pr_{3,3,6}(c),Pr_{5,5,6}(c)),0) : m;p],
         s) ,

Let Bgt: Mpum → Mpu  where  Bgt(a,b,c,i,p,s,m) ≡
        (a,b,c,i,[Eq_1(Or_1(Pr_{4,4,6}(c),Eor_1(Pr_{3,3,6}(c),
         Pr_{5,5,6}(c))),0) : m; p],s) ,
```



```
Let Bhi: Mpum → Mpu  where  Bhi(a,b,c,i,p,s,m) ≡
        (a,b,c,i,[Eq_1(Or_1(Pr_{4,4,6}(c),Pr_{6,6,6}(c)),0) : m; p],
         s) ,

Let Ble: Mpum → Mpu  where  Ble(a,b,c,i,p,s,m) ≡
        (a,b,c,i,[Eq_1(Or_1(Pr_{4,4,6}(c),Pr_{6,6,6}(c)),0) : m; p],
         s) ,

Let Bls: Mpum → Mpu  where  Bls(a,b,c,i,p,s,m) ≡
        (a,b,c,i,[Eq_1(Or_1(Pr_{4,4,6}(c),Pr_{6,6,6}(c)),1 : m; p],
         s) ,

Let Bmi: Mpum → Mpu  where  Bmi(a,b,c,i,p,s,m) ≡
        (a,b,c,i,[Eq_1(Pr_{3,3,6}(c),1) : m; p],s) ,

Let Bne: Mpum → Mpu  where  Bne(a,b,c,i,p,s,m) ≡
        (a,b,c,i,[Eq_1(Pr_{4,4,6}(c),0) : m; p],s) ,

Let Bvc: Mpum → Mpu  where  Bvc(a,b,c,i,p,s,m) ≡
        (a,b,c,i,[Eq_1(Pr_{5,5,6}(c),0) : m; p],s) ,

Let Brs: Mpum → Mpu  where  Brs(a,b,c,i,p,s,m) ≡
        (a,b,c,i,[Eq_1(Pr_{5,5,6}(c),1) : m; p],s) ,

Let Bpl:  Mpum → Mpu  where  Bpl(a,b,c,i,p,s,m) ≡
        (a,b,c,i,[Eq_1(Pr_{3,3,6}(c),0) : m; p],s) ,

Let Bsr: Mpum → Mpu  where  Bsr(a,b,c,i,p,s,m) ≡
        Wr(Wr(a,b,c,i,m,Pd_2(s),Pd_1(s),Pr_{1,8,16}(p),s,
         Pr_{9,16,16}(p)) ,
```

```
Let Jmp: Mpum → Mpu  where  Jmp(a,b,c,i,p,s,m) ≡
          Bra(a,b,c,i,p,s,m) ,

Let Jsr: Mpum → Mpu  where  Jsr(a,b,c,i,p,s,m) ≡
          Bsr(a,b,c,i,p,s,m) ,

Let Nop: Mpu → Mpu  where  Nop(a,b,c,i,p,s) ≡ (a,b,c,i,p,s),

Let Rti: Mpu → Mpu  where  Rti(a,b,c,i,p,s) ≡
          (Read(Sc3(s)),Read(Sc2(s)),Pr3,8,8(Read(Sc1(s)),
           Jn8,8(Read(Sc4(s)),Read(Sc5(s))),Jn8,8(Read(Sc6(s)),
           Read(Sc7(s))),Sc7(s)) ,

Let Rts: Mpu → Mpu  where  Rts(a,b,c,i,p,s) ≡
          (a,b,c,i,Jn8,8(Read(Sc1(s)),Read(Sc2(s))),Sc2(s)),

Let Swi: Mpu → Mpu  where  Swi(a,b,c,i,p,s) ≡
          Stck(a,b,Jn1,1,4(Pr1,1,1(c),1,Pr3,3,6(c)),i,
           Jn8,8(Read(1i13<,1>,0,1,0),Read(1j13<,1>,0,1,1)),
           s) ,

Let Stck: Mpu → Mpu  where  Stck(a,b,c,i,p,s) ≡
          Wr(Wr(Wr(Wr(Wr(Wr(a,b,c,i,p,Pd7(s),Pd6(s),
           Jn1,1,6(0,0,c)),Pd5(s),b),Pd4(s),a),Pd3(s),
           Pr1,8,16(i)),Pd2(s),Pr9,16,16(i)),Pd1(s),
           Pr1,8,16(p)),s,Pr9,16,16(p)) ,
```



```
Let Wai: Mpu → Mpu  where  Wai(a,b,c,i,p,s) ≡
          Stck(a,b,Jn1,1,4(Pr1,1,6(c),1,Pr3,3,6(c)),i,
           [Eq1(Pr2,2,6(c),1) : Pr1,16,17(Jn8,8,1(Read
           (1i14<,1>,0,0),Read(1j14<,1>,0,1),XC3(0)));
           Pr1,16,17(Jn8,8,1(Read(1k12<,1>,0,1,0,0),Read
           (1ℓ12<,1>,0,1,0,1),XC4(0)))],s) ,

Let Clc: Mpu → Mpu  where Clc(a,b,c,i,p,s) ≡
          (a,b,Jn5,1(Pr1,5,6(c),0),i,p,s) ,

Let Cli: Mpu → Mpu  where  Cli(a,b,c,i,p,s) ≡
          (a,b,Jn1,1,4(Pr1,1,6(c),0,Pr3,6,6(c)),i,p,s) ,

Let Clv: Mpu → Mpu  where  Clv(a,b,c,i,p,s) ≡
          (a,b,Jn4,1,1(Pr1,4,6(c),0,Pr6,6,6(c)),i,p,s) ,

Let Sec: Mpu → Mpu  where  Sec(a,b,c,i,p,s) ≡
          (a,b,Jn5,1(Pr1,5,6(c),1),i,p,s) ,

Let Sei: Mpu → Mpu  where  Sei(a,b,c,i,p,s) ≡
          (a,b,Jn1,1,4(Pr1,1,6(c),0,Pr3,6,6(c)),i,p,s) ,

Let Sev: Mpu → Mpu  where  Sev(a,b,c,i,p,s) ≡
          (a,b,Jn4,1,1(Pr1,4,6(c),1,Pr6,6,6(c)),i,p,s) ,

Let Tap: Mpu → Mpu  where  Tap(a,b,c,i,p,s) ≡
          (a,b,Pr3,8,8(a),i,p,s) ,

Let Tpa: Mpu → Mpu  where  Tpa(a,b,c,i,p,s) ≡
          (Jn1,1,6(0,0,c),b,c,i,p,s)
```

5.5 Memory Processors

5.5.1 Overview

The functional specifications for the ROM and RAM processors are short and easy to understand in comparison with those for the MPU. Each memory processor performs a system step every time the MPU performs a read or write operation, regardless of which processor is selected (via address) by the MPU (see the definitions of Read and Write with the definition of Sumpu). Every read or write operation is accompanied by the exchange of 17 bits of information between the MPU and every other processor. The first bit is 0 for a read and 1 for a write; the remaining 16 bits constitute the address. In our particular configuration if the first address bit is 0 then the PIA is selected. On the other hand, if the first two address bits are (1,0) then the RAM is selected; otherwise the ROM is selected by (1,1). Thus there remain 14 bits for addressing within both the RAM and the ROM for a total of 16,384 bytes in each. In the next section (5.5.2) the successor function for the RAM is $Sumem_0$ and for the ROM is $Sumem_1$. The function $Exchmem_0$ decides whether a cell in the RAM is being addressed, and if so, then selects the addressed cell n. Its associated function $f_{0,n}$ then performs a read or write according to the read/write bit. The definition for the ROM is parallel

to that for the RAM, of course, with the exception that the contents of the memory cells are never altered.

5.5.2 Memory Processor Definitions

Let Memspace $\equiv$ $\{{}_0i_{16383}{<},i{>}\}$,

Let Memmap:Memspace $\rightarrow$ V_{14} where Memmap(x) $\equiv$

$[{}_1i_{16383}{<};Eq_1(x,j):Pr_{3,16,16}(Sc_i({}_1j_{16}{<},0{>}))>;$

$({}_1k_{14}{<},0{>})]$,

Let Memory $\equiv$ ${}_0i_{16383}{<}\times V_8{>}$,

${}_0n_1{<}$Let $Sumem_n$:Memory $\rightarrow$ Memory where

$Sumem_n({}_0i_{16383}{<},M_i{>}) \equiv Exchmem_n(XC_5({}_1j_{17}{<},0{>}),$

$({}_0k_{16383}{<},M_k{>}))$,

Let $Exchmem_n$:$V_{17}\times$Memory $\rightarrow$ Memory where

$Exchmem_n(x,Mem) \equiv [Neq_2(Pr_{2,3,17}(x),(1,n)):Mem;$

${}_1j_{16383}{<};Eq_{14}(Pr_{4,17,17}(x),Memmap(j)):f_{n,j}(x,Mem)>;$

$f_{n,0}(x,Mem)]$ replacing "Mem" by "$({}_0i_{16383}{<},M_i{>})$">,

${}_0j_{16383}{<}$,Let $f_{0,j}$:$V_{17}\times$Memory $\rightarrow$ Memory where

$f_{0,j}(x,({}_0i_{16383}{<},M_j{>})) \equiv ({}_0k_{j-1}{<},M_k{>},$

$[Eq_1(Pr_{1,1,17}(x),1):XA_8({}_1\ell_8{<},0{>});$

$Pr_{1,8,16}(Jn_{8,8}(M_j,XA_8(M_j)))],{}_{j+1}m_{16383}{<},M_m{>})$,

Let $f_{1,j}$:$V_{17}\times$Memory $\rightarrow$ Memory where

$f_{1,j}(x,({}_0i_{16383}{<},M_i{>})) \equiv ({}_0k_{j-1}{<},M_k{>},$

$Pr_{1,8,16}(Jn_{8,8}(M_j,XA_8(M_j))),{}_{j+1}m_{16383}{<},M_m{>})>$

5.6 The Peripheral Interface Adapter

5.6.1 Overview

The PIA is intermediate in complexity between the memory processors and the central processor. Most of this complexity is beyond the scope of the present discussion, and it is not to be expected that the definitions for the PIA in section 5.6.2 be comprehensible without a long digression into control signals, individual bits in control registers, and so on. The important points that we wish to raise about the PIA were illustrated in figure 5-1. The figure shows that the state successor function for the PIA is constrained to be evaluated at least as often as the MPU performs a read or write operation. This constraint arises from the fact that the PIA can communicate asynchronously with external devices and so may be ready to transmit interrupt signals before the beginning of a new read or write operation by the MPU. Furthermore, the PIA must operate unhindered even if interrupts are disabled by the MPU. Note that the only exchange function in the definition of the PIA which is not immediate is XA_8 (for the channel DATA of figure 5-1). This exchange corresponds to the transfer of data during a single read or write operation by the MPU whenever the PIA is selected. For a more detailed description of the PIA see [M75].

In the definitions which follow in section 5.6.2, C_0 refers to control register A, D_0 to data direction register A, and P_0 to output register A. C_1, D_1, and P_1 are similarly defined for the B side of the PIA. Note that although the A and B sides of the PIA are alike in many respects, slight differences cause the definition of Supia to be asymmetric with respect to these two sides.

5.6.2 Peripheral Interface Adapter Definitions

Let Pia $\equiv {}_1i_6\langle\times V_8\rangle$,

Let Supiaint:Pia → Pia where Supiaint$(C_0,D_0,P_0,C_1,D_1,P_1) \equiv$

$[Eq_1(Or_1({}_0j_1\langle,And_1(Or_1(Pr_{1,1,8}(C_j),Pr_{2,2,8}(C_j)),$

$Pr_{8,8,8}(C_j))\rangle,1):XS_4(1);XS_4(0)]$,

Let Supia:Pia$\times V_{17}$ → Pia where

Supia$((C_0,D_0,P_0,C_1,D_1,P_1),M) \equiv$

$[Eq_1(Pr_{2,2,17}(M),1):[Eq_1(XS_2(0),1):$

$({}_0i_1\langle,Jn_{1,1,6}(0,0,Pr_{3,8,8}(C_i)),D_i,P_i\rangle);$

$({}_0j_1\langle,Jn_{1,1,6}(Or_1(Pr_{1,1,8}(C_j),[Eq_3(Pr_{3,5,8}(C_j),(1,0,0)):$

$[Eq_1(XS_{j,1}(Comp_1(Pr_{7,7,8}(C_j))),Pr_{7,7,8}(C_j)):$

$Pr_{1,1,2}(1,XS_{j,2}(1));0];$

$[Equiv(XS_{j,1}(Comp_1(Pr_{7,7,8}(C_j))),Pr_{7,7,8}(C_j))]),$

$Or_1(Pr_{2,2,8}(C_j),[Eq_1(Pr_{3,3,8}(C_j),1):0;$

$Equiv(XS_{j,2}(Comp_1(Pr_{4,4,8}(C_j))),Pr_{4,4,8}(C_j))]),$

$Pr_{3,8,8}(C_j)),D_j,P_j\rangle)];$

$[{}_0k_1\langle;Eq_2(Rsel,(k,0)):Eq_1(Pr_{6,6,8}(C_k),0):$

$[Eq_1(Rw,0):Wrpia(Regs,XA_8(D_k));Rdd_k(Regs)];$

$[Eq_1(Rw,0):Wrp_k(Regs,$

$XA_8(Or_8(And_8(XS_{k,3}(X)),Comp_8(D_k)),And_8(P_k,D_k))));$

$Rdp_k(Regs)];$

$Eq_2(Rsel(k,l):[Eq_1(Rw,0):Wrpia(Regs,XA_8(C_k)];$

$Rdc_k(Regs,X)>;(Regs)]$

replacing "Regs" by "C_0,D_0,P_0,C_1,D_1,P_1", replacing "Rw" by "$Pr_{1,1,17}(M)$", replacing "Rsel" by "$Pr_{16,17,17}(M)$", replacing "X" by "${}_1m_8<,0>$",

Let $Wrpia:Pia\times V_8 \to Pia$ where $Wrpia(R,S) \equiv R$,

Let $Rdd_0:Pia \to Pia$ where $Rdd_0(C_0,D_0,P_0,C_1,D_1,P_1) \equiv$

$(C_0,XA_8({}_1i_8<,0>),P_0,C_1,D_1,P_1)$,

Let $Rdd_1:Pia \to Pia$ where $Rdd_1(C_0,D_0,P_0,C_1,D_1,P_1) \equiv$

$(C_0,D_0,P_0,C_1,XA_8({}_1i_8<,0>),P_1)$,

Let $Wrp_0:Pia\times V_8 \to Pia$ where

$Wrp_0((C_0,D_0,P_0,C_1,D_1,P_1),S) \equiv (Jn_{1,7}(0,Pr_{2,8,8}(C_0)],$

$D_0,Pr_{1,8,9}(Jn_{8,1}(P_0,[Eq_3(Pr_{3,5,8}(C_0),(1,0,0)):XS_{0,2}(0);$

$0])),C_1,D_1,P_1)$,

Let $Wrp_1:Pia\times V_8 \to Pia$ where

$Wrp_1((C_0,D_0,P_0,C_1,D_1,P_1),S) \equiv (C_0,D_0,P_0,$

$Jn_{1,1,6}(Pr_{1,1,8}(C_1),0,Pr_{3,8,8}(C_1)),D_1,P_1)$,

Let $Rdp_0:Pia \to Pia$ where $Rdp_0(C_0,D_0,P_0,C_1,D_1,P_1) \equiv$

$(C_0,D_0,XA_8({}_1i_8<,0>),C_1,D_1,P_1)$,

Let $Rdp_1:Pia \to Pia$ where $Rdp_1(C_0,D_0,P_0,C_1,D_1,P_1) \equiv$

$(C_0,D_0,P_0,C_1,D_1,XA_8({}_1i_8<,0>))$,

Let $Rdc_0:Pia\times V_8 \to Pia$ where

$Rdc_0((C_0,D_0,P_0,C_1,D_1,P_1),S) \equiv (Jn_{1,1,8}(Pr_{1,1,8}(C_0),$

$[Eq_1(Pr_{3,3,8}(S),0):Pr_{2,2,8}(C_0);0],$

$Pr_{3,8,8}([Eq_2(Pr_{3,4,8}(S),(1,1)):Pr_{1,8,9}(Jn_{8,1}(S,$

$XS_{0,2}(Pr_{5,5,8}(S)))];S])),D_0,P_0,C_1,D_1,P_1)$,

Let $Rdc_1:Pia\times V_8 \to Pia$ where

$Rdc_1((C_0,D_0,P_0,C_1,D_1,P_1),S) \equiv (C_0,D_0,P_0,Jn_{1,1,8}(Pr_{1,1,8}(C_1),$

$[Eq_1(Pr_{3,3,8}(S),0):Pr_{2,2,8}(C_1);0],$

$Pr_{3,8,8}([Eq_2(Pr_{3,4,8}(S),(1,1)):Pr_{1,8,9}(Jn_{8,1}(S,$

$XS_{1,2}(Pr_{5,5,8}(S)))];S])),D_1,P_1)$,

Let $XS_7:V_{17} \to V_{17}$,

Let $XA_8:V_8 \to V_8$,

Let $XS_2:V_1 \to V_1$,

Let $XS_3:V_1 \to V_1$,

${}_0i_1<,{}_1j_3<$, Let $XS_{i,j}:V_1 \quad V_1>>$

6. Conclusions and Future Work

We now present a brief summary of the major conclusions we have drawn from the results in this report. That summary will be followed by a characterization of our immediate future plans in the form of a technical proposal (submitted in response to RFQ DASG60-78-Q-0062) for further work.

6.1 Conclusions

The major conclusions are based on section 3, "A Theory of Design." The present work is sufficient to demonstrate, at least in theory, the possibility of using such formal methods in system development. If the theory developed is validated by practice, it would be possible to build a computer assisted formal engineering laboratory that would have significant impact on the desired payoffs for large scale real time system developments.

The research method developed as part of this work is generally applicable and has succeeded in producing a basic design theory for distributed data processing systems. It describes an iterative procedure that has been applied once in this work. Future work will involve extension of the formally treated properties and an iteration of the research procedure.

The potential direct payoffs of a DDP development system are:

-- predictability
-- generality
-- reliability
-- simplicity
-- efficiency
-- testability of specifications
-- testability of realizations

The achievement of these payoffs can be significantly more complicated in a DDP design. Thus a serious need for a design theory in centralized data processing becomes an absolute necessity for DDP. The design theory reported here is a significant advance in both our understanding of DDP development systems and our ability to achieve the above payoffs.

We have developed a formally defined set of properties for formal specifications. The presence of these properties makes it possible to support a formal design methodology. Their absence severely constrains the scope of design analysis and specification tools. The set of properties is as follows:

-- formal
-- consistent
-- algorithmic
-- homogeneous
-- hierarchically modular
-- extensible
-- distributed
-- complete

These properties may be used to compare and classify design methodologies from the design automation point of view.

Following our research procedure, we next developed a formal distributed system specification language that has all of the formal properties mentioned above. The language is essentially normal algebraic notation interpreted as functional process specifications. The introduction of formal "exchange functions" allows the algebraic notation to be used for asynchronously interacting processes at a functional level of abstraction. Such specifications are designed to maximize the applicability of automated analysis and transformation tools. We can thus provide automated analysis and validation aids not previously possible.

We have developed a set of specification transformation procedures that can be substantially automated and can be automatically validated. These procedures include design steps of:

-- approximation
-- decomposition/integration
-- elaboration
-- optimization
-- evolution.

These steps are sufficient to support an entire DDP system development process. Others will be added as this work continues.

We have developed some design principles that will guide a designer in the application of our methodology. These principles are only a beginning and have not been tested in practice.

6.2 Future Work

The following technical proposal was prepared in response to RFQ DASG-60-78-Q-0062 and describes our immediate future plans for the continuation of this work.

6.2.1 Research Area

The System Development System (SDS) developed by BMDATC was largely designed and implemented in the context of centralized data processing systems. SDS represents the current state of the art for development systems.

The introduction of Distributed Data Processing (DDP) concepts into BMD systems produces a need to deal explicitly with the additional complexities of asynchronous interactions and the greater range of potential solution architectures justify a fresh look at SDS methodology. The goals are to identify and design suitable extensions required to support DDP development.

This work will start with the initial critical area of requirements specifications as represented by the Requirement Specification Language (RSL) component of SDS.

6.2.1.1 Background (SOW 1.0)

The previous work on BMDATC Contract DASG60-76-C-0080 has laid a theoretical foundation for significant improvements in formal DDP system specification, analysis, and development. This foundation includes a set of formally defined DDP system specification language properties, a set of analysis tools to determine the presence of these properties in a given specification, an automatically generated simulation of specifications having the specified properties, and a set of design transformations sufficient to support a DDP development process while ensuring the invariance both of these properties and of previous design decisions during the development process.

This proposal is to apply this theoretical foundation to extending RSL to include more useful DDP concepts and increasing the scope of automatable analysis and simulation tools for RSL specifications. The following properties have been formally defined and are required as a basis for the desired RSL extensions. These properties may be extended as necessary in the course of this contract. An informal characterization of them is given below. A formal characterization will be found in the final report of the current research contract.

The changes made to the previous proposal and incorporated in this revised proposal were to prioritize the research requirements for a "level of effort" approach.

6.2.1.1.1 Formal

A specification is formal if it is abstract and decidable that it is a specification. The relevant properties of the abstraction are precisely stated and potentially susceptible to automated analysis and transformation.

Abstractions also provide us a "transparency" for our design theory. We will incorporate only the properties of the abstraction into our formal design theory. The remaining properties will be neither treated nor prejudiced by our design theory. Where our design theory does not help, it also does not hinder.

There are a large number of automatable analysis possible because of this property alone. For example, a "specification" can be syntactically checked by a parsing algorithm to decide that it is completely specified and correctly formed. A designer could thus obtain the type of feedback currently available to a programmer from a compiler.

6.2.1.1.2 Consistent

A specification is consistent if it is interpretable as a system. A consistent specification thus has no internal contradictions. The arguments of each function are in the functions domain and an implementation of the specifications could be constructed.

Clearly if the formal specifications include performance data, the task of formally proving consistency may be impossible

in the current state of the art. Logical consistency (non-performance properties) may be the best we can do at present. Even this restricted form is of great utility in producing specification of large scale and complex systems.

6.2.1.1.3 Algorithmic

A specification is algorithmic if, for a given state of the specified system, we can automatically generate the set of possible immediate successors to that state.

In effect, an algorithmic specification need not be procedural but it must be automatically interpretable. It is thus its own simulation model. With this property we may display the behavior of the specified system to the level at which it has been specified.

A designer may thus automatically obtain semantic feedback as to the actual consequences of a design decision by simulating desired test cases, just as a programmer can make test runs on a program via a compiler.

6.2.1.1.4 Homogeneous

A specification language is homogeneous if there is a specification for each level of abstraction (generality) of the specified system. A homogeneous specification language thus provides a way to specify design decisions in any phase of the development process.

The homogeneous property not only makes the corresponding design methodology applicable to the entire development process but also makes it possible to provide the continuity required to preserve the validity of earlier design decisions in subsequent phases.

6.2.1.1.5 Hierarchically Modular

Modularity is equivalent to a "local homogeneity." Design of a module can also be carried out and tested independently. Each module may also be decomposed into more primitive modules thus forming a hierarchy.

This property is required in order to factor the complexity of a specification, and is quite similar to the concepts of structured programming applied to design of systems.

6.2.1.1.6 Extensible

A specification is extensible if it provides a way to include uninterpreted (informal) attributes that convey the required information but are not formally included in the specification abstraction. At the current state of the art, such attributes will include most performance requirements.

We must not hinder, by our formalisms, the ability of designers to do informal analysis. Without this property the specification language would prevent a designer from doing many things that must be done but that cannot at present be automatically analyzed.

6.2.1.1.7 Distributed

A specification must be applicable to distributed systems. The interpretation of a specification must be as an explicitly specified distributed system. An implicit specification of DDP attributes makes them inaccessible to automated analysis, both syntactic and semantic. The designer requires feedback from such analysis when dealing with the system attributes uniquely related to DDP. The designer must also be able to make and enforce decisions about distribution and interactions at appropriate levels of abstraction (generality).

Distributed specifications must thus be based on some formal model of asynchronously interacting subsystems, in which both the distributional and the interactional attributes are accessible for analysis and simulation.

6.2.1.2 Objectives (SOW 2.0)

6.2.1.2.1 Long Term Objectives

The long term objectives of this work are to produce a practical science of DDP design, a theory of development processes for DDP systems, and a methodology including a substantial and practical set of automated tools.

In effect, we want to generalize on the concepts of structured programming to produce concepts of structured design, while retaining the programmers' ability to analyze a specification both syntactically and semantically. Further, we want to

structure the development process so that the consistency of a specification and the validity of previous design decisions can be automatically verified.

6.2.1.2.2 Short Term Objectives

The short term objectives are to analyze and extend RSL to incorporate as many of the desirable properties as possible, and to extend the domain of formal attributes to which the properties are applied. The results of this short term work will be used to specify some test cases in the ongoing BMD DDP integrated experiment program.

6.2.1.3 Research Requirements (SOW 3.0)

The theoretical base developed under contract DASG60-76-C-0080 through modification P00004 will be used to accomplish the research requirements. Some modifications or extensions to that previous work may also be required.

The research requirements will be addressed on a "level of effort" basis with priorities defined by the following order. Each subsequent requirement is dependent on resolution of the critical issues of the preceding ones.

6.2.1.3.1 Specification Properties (SOW 3.1)

The first order set of specification properties developed in previous work will be defined and extended as required through findings during this contract period. The resulting set should be directly relevant to DDP system development.

6.2.1.3.2 Analysis of BMDATC Requirement Statement Language (SOW 3.2)

The RSL component of SDS will be analyzed to determine the domains of system attributes, specifiable in RSL, for which the required formal properties (described in Section 1.1) are testable. These properties and their associated domains will be reported in terms of RSL concepts.

A plan both for extending RSL to include properties shown to be not present in the analysis above and for enlarging the corresponding domains of system attributes will be prepared.

6.2.1.3.3 Design Language Extensions (SOW 3.3)

The recommended extensions to RSL (from 6.2.1.3.2) will be designed. These extensions will include not only the RSL specification language but also the associated automated analysis and simulation tools.

Assistance in the implementation of the extensions will be provided. The validation and verification of the resulting implementation will be provided.

6.2.1.3.4 DDP Experiments

A formal specification, exploiting the extended RSL language, for the BMD DDP integrated experiment program at a suitable level of design will be prepared to demonstrate the effectiveness of the language modifications.

The usefulness of such a specification for an analysis of the specified system will be shown by example.

6.2.2 Overall Approach

The work is clearly experimental and of a research nature. The actual sequence of the approach may be strongly affected by the nature of the intermediate results. The following is a base line approach to be modified as required to meet the contract objectives.

6.2.2.1 RSL Analysis (SOW 3.2)

We will first study RSL as it is and describe a suitable formal model for it. We will validate that model via the current RSL implementation. We will then find, in terms of that model, the applicable domains for testability of each of the required formal properties. We will then identify extensions required for DDP applications and prepare a plan for making those extensions.

6.2.2.2 RSL Extensions (SOW 3.3)

We will design the planned extensions and specify them formally. We will assist BMDATC selected contractors in the implementation of the specified extensions.

We will develop and carry out a test plan for the validation and verification of the implementations.

6.2.2.3 DDP Experiments (SOW 3.4)

We will select an appropriate phase in the BMDATC DDP integrated experiment program to apply and demonstrate the

usefulness of the extended RSL specifications. To the extent possible by the state of the implementation we will conduct automated analysis of the resulting specification for essential DDP properties. We will thus identify deficiencies in the RSL extensions.

6.2.2.4 Specifications (SOW 3.1)

We will address the deficiencies discovered above by modifying and augmenting our formal properties. We will then prepare a plan for further research for RSL improvements.

6.2.3 Critical Issues

6.2.3.1 RSL Analysis (SOW 3.2)

The critical issue for RSL analysis is to obtain a validated formal model for RSL specifications and their interpretation in terms of systems.

The approach to be taken will be to postulate a suitable formal system domain and establish the interpretation relation between RSL specifications and members of that system domain. The validity of the interpretation will be tested by tests of examples of RSL constructs.

6.2.3.2 RSL Extension (SOW 3.3)

The critical issues seem to be the domain where consistency is testable and how to extend to describe explicitly the interactions between asynchronous systems.

The approach to the consistency issue will probably take the form of testable constraints on the usage of RSL constructs. The interactions between R-nets for interacting systems will have to be explicitly represented. We will try to accomplish this by the introduction of the exchange functions developed in our current contract work.

6.2.3.3 DDP Experiments (SOW 3.4)

The first critical issue is the selection of the phase to use for our demonstration.

The approach will be to select that phase in which interactions, synchronization, and control will be specified and analyzed.

The second critical issue is how to carry out the demonstration of the power of our extensions and associated automated tools when their implementation may not be completed. It is very difficult to predict the timing of their availability in a "level of effort" contract that is also dependent on other BMDATC contractors for the implementation.

If required, the specifications will be developed and analyzed "by hand," although the correctness of such specifications and analysis cannot be guaranteed without the proposed automated tools. At the least, an illustration of the attainable results will be provided.

6.2.3.4 Specifications (SOW 3.1)

The critical issues in identifying and defining valuable new properties for RSL extensions involve the inclusion of performance constraints into the formal models.

The approach will be to identify some critical dynamic analysis problems and find sufficient design constraints to extend formal consistency to include the required performance attributes.

6.2.4 Work Plans

We plan to carry out the work at the University of Wisconsin utilizing the normal facilities and equipment of the university. We will require access to and use of the ARC computers as described below. No subcontractors or consultants will be employed.

6.2.4.1 Research Effort

The proposed work is experimental in nature and will be pursued as a "level of effort" contract. The proposed level of effort in the accompanying cost quotation should be sufficient to accomplish the short term objectives applied to the required research.

The staffing during this spring semester will be limited by the mid-semester availability of qualified candidates. Maximum staffing should be reached this summer, coincidental with the decomposition and elaboration of our research approach.

The majority of the staff will be highly qualified graduate students working on (or preparing for) Ph.D. theses in the area of this contract. The quality of work will be such that the major results will be published in appropriate technical journals and meetings.

I plan to work on this contract full time this summer and half time during the next fall and spring semesters.

Some clerical support for report preparation, program preparation, etc., will also be required.

6.2.4.2 Workshops

We propose to conduct, as part of the required oral reports, several DDP design workshops for both BMDATC personnel and their contractors. The workshops will be designed to communicate the extended RSL methodology and to obtain feedback on the proposed extensions and their use.

6.2.4.3 Computer Usage

We will use the ARC computers to study and validate models of RSL interpretation. We will also require ARC computers to implement and test the RSL extensions.

We will access ARC facilities via university terminals over DAIN lines and via on-site visits.

We will develop and test algorithms using the university computers so as to minimize the required visits to the Huntsville, Ala. ARC facility.

We estimate that about ten hours of CDC 7600 time at ARC would be reasonable for our participation, but this is a very rough estimate.

Appendix A - Specification of Asynchronous Interactions Using Primitive Functions

This appendix consists of the original unedited text of a paper by Pamela Zave and D. R. Fitzwater. The original page numbering can be found in the upper right hand corner of each page. Page numbering for the present report continues uninterrupted at the bottom center of each page.

SPECIFICATION OF ASYNCHRONOUS INTERACTIONS

USING

PRIMITIVE FUNCTIONS

by

Pamela Zave
Department of Computer Science
University of Maryland
College Park, Maryland 20742

and

D. R. Fitzwater*
Department of Computer Sciences
University of Wisconsin
1210 W. Dayton Street
Madison, Wisconsin 53706

* Supported in part by BMDSC (ATC-P) Contract No. DASG 60-76-C-0080.

SPECIFICATION OF ASYNCHRONOUS INTERACTIONS
USING PRIMITIVE FUNCTIONS

Abstract: Three primitive functions appear to be sufficient to specify all useful asynchronous interactions. They can be used for requirement and design specification at all levels of abstraction. The resulting specifications are shown to be well suited to analysis.

I. INTRODUCTION

Much attention is now being given to the early stages of the construction of digital systems: requirements analysis and system design ([Ro]). Continuation of this work requires formal representations for systems, so that automated design and analysis tools can be developed.

The most important characteristic of a suitable representation would be to specify properties of systems effectively, while placing as few constraints as possible on unspecified properties. If this could be done at any level of abstraction (i.e. using any sufficient set of primitives), then the representation would facilitate decomposition of the design problem, and composition of the solutions to sub-problems. Given the multiplicity of logical, resource, and performance requirements to be satisfied, this appears to be the most promising approach.

This paper proposes a solution to the problem of specifying asynchronous interactions between processes without having to specify data and control structures - yet in such a way that the represented design can be analyzed and tested.

The key to specifying digital system properties without premature binding of other properties is use of the mathematical function, the most abstract formal characterization of computation known. A single process can be represented functionally as a pair (Σ,f), where Σ is a state space and f is a successor function on Σ. A computation of this process is a sequence $\sigma_0,\sigma_1,\sigma_2,\ldots$ such that $\sigma_i \in \Sigma$, $i \geq 0$, and $\sigma_{i+1} = f(\sigma_i)$, $i \geq 0$. A digital system, in general, can be represented as a set of interacting processes which run asynchronously.[1] A computation of such a system

[1] There are other ways to use functions to specify systems, but this seems the most useful. We could use a function to characterize the behavior seen by an external observer, for instance, but it would vary with the observer.

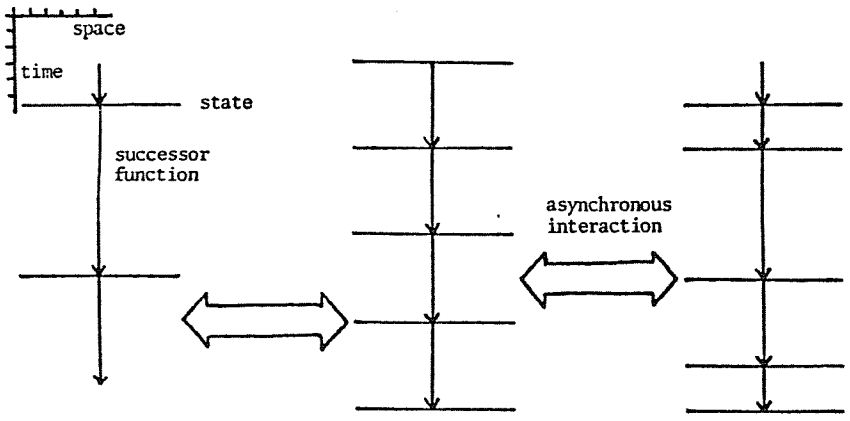


Figure 1. Space-time diagram of a digital system.

can be viewed as in Figure 1.

In the next section three primitive functions will be defined. They seem sufficient for specifying all useful asynchronous interactions, as will be substantiated by examples. These functions are to be used as sub-functions of the successor function, and are therefore compatible with any notation in which primitive functions can be introduced.

Section III shows that specifications using these functions are well suited to analysis. Axiomatic proof techniques and simulation are both applicable. Properties such as mutual exclusion, termination, and freedom from deadlock can often be decided from efficient syntactic analysis.

Section IV discusses the relation of these results to other work, and plans for future work.

II. THE FUNCTIONS AND THEIR GENERALITY

A. The Functions

The three primitives are called collectively exchange functions.

Consider two processes which operate in (mutually synchronized) master-slave mode.

Their communication can be specified using the XC (eXchange to Communicate) function. At the beginning of its step (the computation of the successor function), the slave process initiates evaluation of XC('ok'). If the master has no command ready, evaluation of XC('ok') waits. When the master has a command ready, it initiates XC(< command >), upon which evaluations of both functions can terminate. They exchange arguments, so that the XC in the slave returns the command (which is executed in the rest of the slave's process step), and the XC in the master returns 'ok', indicating that the command was given to a healthy slave.

If the master initiates another XC (< command >) before the slave is finished, it will have to wait until the slave finishes its process step and initiates a matching XC. These interactions are illustrated in Figure 2.

Every exchange function belongs to a class (denoted with a subscript), and can only exchange with other members of its class. Thus the master "owns" the slave because only the master executes exchanges in the slave's class (although who is

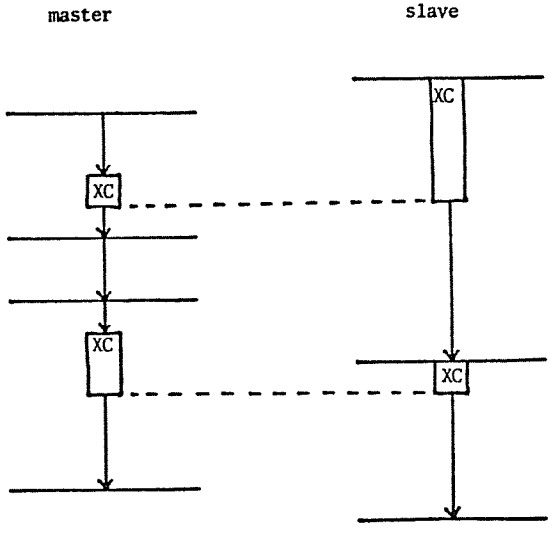


Figure 2. Master-slave communication.

master is strictly an interpretation of the messages exchanged).

Now consider a process which represents execution of instructions in a computer. The interrupt mechanism can be specified using XC and the XS (eXchange Synchronous) function. The device causing the interrupt initiates an XC_i(< interrupt condition>). At the end of each process step, i.e. instruction execution, the computer process executes an XS_i (with an argument which cannot be an interrupt message), which behaves like the XC_i except that it will not wait to be matched. If there is an XC_i (interrupt) pending, the two functions will exchange and return. If there is not, the XS_i returns directly with its own argument as its value. The computer process can determine the outcome from the value returned. This interaction is shown in Figure 3.

Finally, consider a set of processes which have need of a real-time clock. The clock itself is a process which "ticks" every time it takes a step. On each step it executes an XS (<current time>), thereby offering the current clock value to any process that wants it.

A process could read the clock by evaluating an XC (argument irrelevant) in the clock's exchange class, but this leads to problems. If two processes ask for the clock value before the clock ticks again, their pending XC's will exchange with each other! To prevent this we introduce a third function, XA (eXchange Asynchronous). XA is the same as XC except that XA's cannot exchange with each other. If the two reading processes use XA's, they will interact with the clock process as shown in Figure 4. Since, in this scheme, two processes can never read the same clock value, real times can be used for conflict resolution.

Figure 5 summarizes the possible interactions between exchange functions in the same class.

The real-time clock example shows that general implementation of these primitives requires conflict resolution in a distributed network, i.e. a means by which individual function evaluations can be matched into interacting pairs. Our only

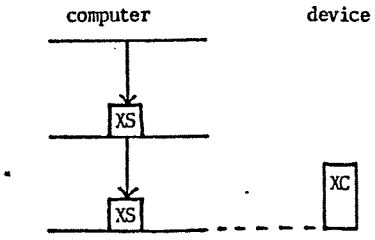


Figure 3. An interrupt.

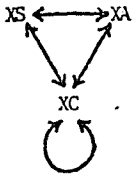


Figure 5. Possible interactions in a class.

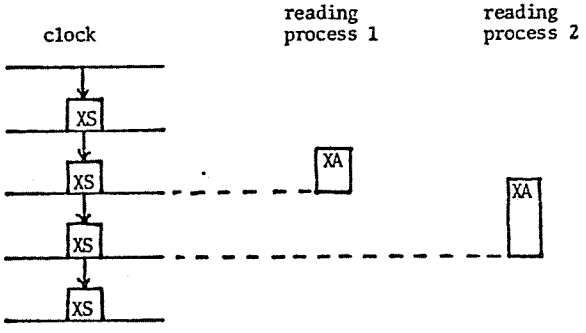


Figure 4. Reading a real-time clock.

logical requirement on matching is that no pending exchange be locked out. In the real time clock example, for instance, this "fair scheduling" rule could be implemented by sending all XA initiations to a queue at the clock's network node and matching the XA's to XS's in First-Come-First-Served order. This matching might not be FCFS in real time because of different transmission delays experienced by the XA initiation messages, but it would prevent lockout. It is also possible to do the matching FCFS in real time, to a known margin of error, as shown in [La2].

Exchanges are exactly like other computable functions in that their evaluation is initiated with an argument, and some time later evaluation terminates, returning a value. Only XC's and XA's can fail to return, and only if they are never matched. Unlike many functions, however, exchanges may be non-deterministic[2], and their evaluation has side effects.

B. Generality of the Primitives: Breadth

In this section we will substantiate the claim that exchanges are sufficient primitives for specifying the whole domain of useful asynchronous interactions, by showing that they can be used to specify a very general message communication facility. In this facility, each process sends zero or one message to each other process at the completion of its step. Messages arriving at a process are queued in arrival order; at the beginning of its step, the process absorbs all the messages which arrived since the last time it began a step. The assumptions made about message transmission are (a) that it takes an arbitrary, finite, non-zero period of time, (b) that messages are not lost, and (c) that messages from one source to one destination arrive in the order they were sent. Those assumptions are commonly made to keep message systems theoretically tractable ([La2], [FK]). A typical process step in this system is illustrated in Figure 6.

As an intermediate step, we define a process which is a producer-consumer buffer. On each step it executes one XS in the producers' class and one XS in the consumers' class; it can thus stay the same, grow by one element, lose one element, or both.

Let:

"BUFFER" be the ordered list which is the state of the process;

null be the null element;

"first" be the function on a list whose value is its first element;

"rest" be the function on a list whose value is everything but the first element;

[2] A non-deterministic function returns a single value chosen non-deterministically from a subset of its range. For some non-deterministic functions (but not exchanges) the subset is a proper subset determined by the argument.

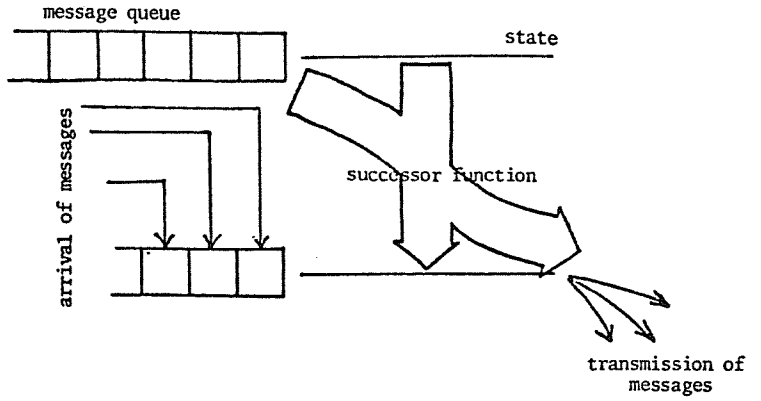


Figure 6. A process step in a message system.

"first-insert (L,e)" be a function whose value is the list L with element e added as its first (or L if e = null)

"last-insert (e,L)" be a function whose value is the list L with element e added as its last (or L if e = null).

Then the process is defined by its successor function:

successor (BUFFER) =

last-insert (XS_p(null),

first-insert(rest (BUFFER), XS_c(first(BUFFER)))),

assuming that producers evaluate XA_p (NEW-ELEMENT) to do input, and consumers evaluate XA_c (null) to do output. Notice that although this functional specification of the buffer is logically complete, it allows deferment of such questions as: "Network or multiprogrammed implementation?" "Many producers and consumers, or only one of each?"

In the functional specification of the message system, each process has an exchange class T_i through which other processes "transmit" to its message queue, which is a producer-consumer buffer similar to the one above. The process also has a

private exchange class U_i through which it communicates with its queue.

If a process wishes to send messages $m_1, m_2, \ldots, m_n$ to processes $P_1, P_2, \ldots, P_n$ at the end of its step, it initiates parallel evaluations of $XA_{T_i}(m_i)$, $1 \le i \le n$, as its next step begins (as part of its successor function). The step cannot end until all XA's have returned, but this prevents messages' being received in scrambled order - since completion of a step with some of the previous step's messages unreceived would mean that newer messages could be received sooner. Note that the transmitter must synchronize itself with the buffer process, but not the receiver.

The successor function of the "message queue" buffer is:

$$\text{successor}'(\text{BUFFER}) = XS_{U_i}(\text{last-insert}(XS_{T_i}(\underline{\text{null}}), \text{BUFFER})).$$

If the process has initiated a step before this buffer process step, its XA_{U_i}(empty-buffer) will capture the entire buffer contents. This occurs in Figure 7.

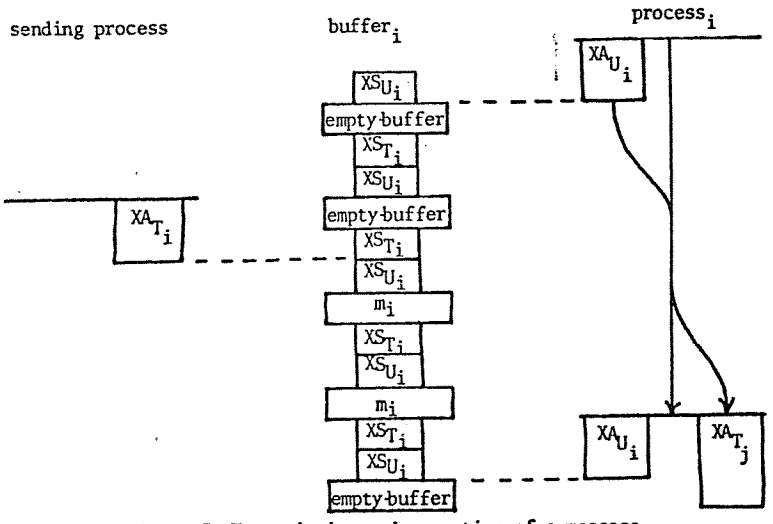


Figure 7. Transmission and reception of a message

The only property of the verbal specification which has not been included in the formal one is the arbitrary transmission delay. It is often not important to do so, because asynchronous processes are by nature designed to be insensitive to relative rates. Let us assume, however, that we are specifying a network with widely separated nodes and severe real-time constraints on communication. Then the transmission delay could be specified by another buffer process, between the sender and the present buffer, with a delaying function. Satisfaction of the real-time constraint then involves (a) ensuring that all function evaluations on the communication path are logically bounded (including the implementation and matching of exchanges), (b) assigning a fraction of the allowable delay to each logical step, and (c) finding an implementation technology which can meet that performance requirement.

C. Generality of the Primitives: Depth

In this section we will substantiate the claim that exchanges are sufficient primitives for specifying asynchronous interactions in digital systems at all levels of abstraction, again by example. We consider specifications at different levels of abstraction to differ in the nature of the properties that are specified.[3]

Our first example will give two specifications of a "toy" missile defense system: the first specifies the problem, or requirement, and the second specifies its solution at a high level.

In the problem specification there are two processes, a digital simulation of a missile and a digital simulation of an interceptor. The missile moves according to its trajectory, but sending its position and velocity to the interceptor as it does so. The interceptor adjusts its own trajectory based on that information, until the two projectiles collide. The problem specification is thus based on perfect knowledge and control; the requirement is to design a system which approximates its

[3] "Level" indicates a linear ordering among abstractions which is not absolutely necessary, but seems to be a characteristic of useful systems designs.

behavior using only approximate knowledge based on ground radar, and remote control of the interceptor.

Let:
"MISSILE" be the position and velocity of the missile;
"move" be the function which calculates the new value of MISSILE after t seconds;
"INTERCEPTOR" be the position and velocity of the interceptor;
"approach (INTERCEPTOR,MISSILE)" be the function whose value is the new position and velocity of the interceptor in t seconds, after correcting for the present state of the missile;
p_i^j be the projection function onto the ith component of a j-tuple.

Then the two processes of the requirement are defined by:

missile-successor (MISSILE) =

p_1^2 (move(MISSILE),XC_I(move(MISSILE)));

interceptor-successor(INTERCEPTOR) =

approach(INTERCEPTOR,XC_I(null)).

A solution to this problem, an implementable approximation, is sketched in Figure 8.

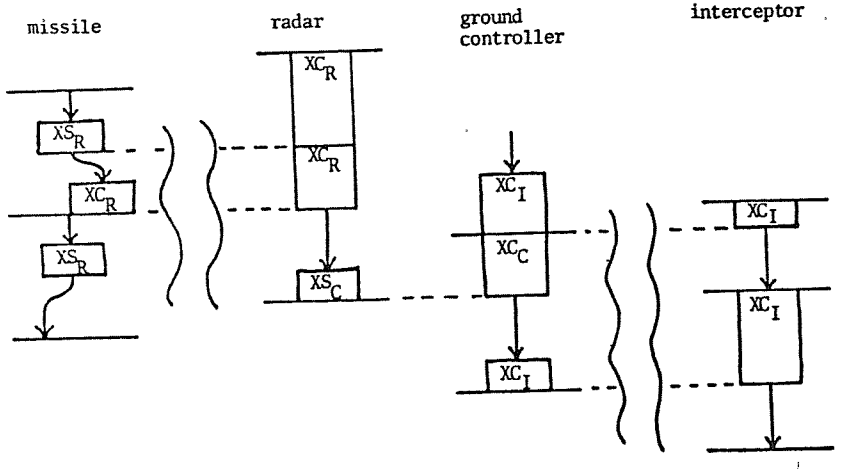


Figure 8. The missile defense system.

The radar process communicates with the missile twice on each step: once in emitting and once in receiving. If the missile process receives emission on a step, it returns it in that step. Whether it reflects radar waves or not, the missile process continues to be a digital simulation of the missile.

When the radar receives its reflected waves (with statistical error introduced), it calculates an approximate position for the missile, which it then relays to the ground controller. The ground controller process combines the new position estimate with its present estimates of the position and velocity of both the missile and the interceptor, and generates commands for the interceptor to follow. These are finally transmitted to the interceptor.

Given this explanation and highly explanatory function names, it is hoped the following definitions will be clear. It is important to note that the successor functions of all four processes interact with real time clocks in a way that would only be made explicit by elaboration (i.e. formal definition) of the functions left as primitives here. Thus MISSILE' and INTERCEPTOR' differ from their previous definitions in that they contain time records, and move' differs from move in that it reads the real time $u+\Delta u$, compares it to its last time reading u, and calculates a movement over time Δu (instead of the constant t used by move). If necessary, the various signal propagation delays could be specified as described for the message system.

```
missile-successor(MISSILE') =
        p1^2(move'(MISSILE'),
           (XSR(null) ≠ null,
              XCR(introduce-error(position-only(move'(MISSILE'))))
           )
        )[4];
```

[4] This uses the LISP-like function $(p_1,g_1;p_2,g_2;\ldots;p_n,g_n)$ where p_i is a predicate and g_i is a function, $1 \le i \le n$, and the value returned is that of the first g_i such that p_i = true.

radar-successor(NOTHING) =

p_1^2 (null, XS_C(calculate-approx-position(XC_R(XC_R(EMIT))))

);

controller-successor(MISSILE-APPROXIMATION, INTERCEPTOR-APPROXIMATION) =

(update-using (XC_C(null)),

simulate-with-error-obedience-of (INTERCEPTOR-APPROXIMATION, "command")

)

where "command" stands for:

p_1^2 (calculate-control

(MISSILE-APPROXIMATION, INTERCEPTOR-APPROXIMATION, XC_C(null)),

XC_I(calculate-control

(MISSILE-APPROXIMATION, INTERCEPTOR-APPROXIMATION, XC_C(null)))

)[5];

interceptor-successor(INTERCEPTOR') =

obey(INTERCEPTOR', XC_I(null)).

Often the successor function of a process is implemented by a set of asynchronously interacting parallel processes, as shown in Figure 9. Exchange functions can also be used to specify the intra-process interactions between these implementation processes. In the interrupt example (Figure 3), for instance, the process steps which represent instruction execution can be implemented by interacting CPU and memory processes. If the memory were dedicated to the processor (as in a minicomputer system without direct-memory-access I/O), then they could communicate using XC's. This would be similar to Figure 2, except that each instruction must be expanded to a pair:

[5] This crude functional notation lacks the crucial ability to specify that the three instances of XC_C all stand for the value returned by a single evaluation.

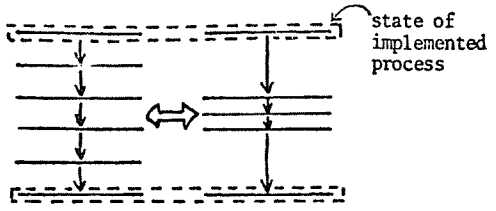


Figure 9. Asynchronous processes implementing a successor function.

one to make a memory reference, and one to return its result.

We should also say something about the opposite case, implementation of asynchronous processes by a single process, as in the multiprogramming implementation of concurrency. The implementing process is the instruction-execution process of Figure 3, and the asynchronous processes are an interpretation of it. Interactions among the asynchronous processes can be specified with exchanges when the processes themselves are being specified, but not in the specification of the multiprogrammed implementation - because there is no longer anything asynchronous or parallel going on. Some instruction (interpreted as belonging to P_1) will write in a memory word, which will later be read by another instruction (interpreted as belonging to P_2). This is actually part of the implementation of an exchange between P_1 and P_2.

III. ANALYSIS OF ASYNCHRONOUS INTERACTIONS

A. Simulation and Testing

At the very least, we must be able to determine whether or not processes interact as they should by simulating or implementing them, and testing their behavior. The presence of exchange functions in specifications does not prevent this, as the functions are well defined and effective. For instance, the missile defense system of Figure 8 would be simulated, with the expected statistical errors introduced, to see if it approximated the requirement specification sufficiently well. In fact, the functional specification is the simulation model.

B. Axiomatic Proof Techniques

Another technique for determining correctness is that of axiomatic proof, as applied to parallel processes in such work as [Ke], [La1], and [OG]. Axiomatic proofs are also compatible with the use of exchange functions.

This will be illustrated using the formalism of [La1], in which a parallel program (process) is represented as a flowchart.[6] An interpretation is an association of a (hopefully) invariant assertion with each arc in the flowchart. The interpretation is consistent if, when the assertion on the input arc of a node is true before execution of the node, the assertion on its output arc must be true after execution. A set of processes is represented as a set of flowcharts, and proofs consist of showing either that (a) the interpretations of all processes remain invariant regardless of the sequence of execution steps (safety properties), or that (b) the truth of one assertion implies that another assertion will eventually become true (liveness properties).

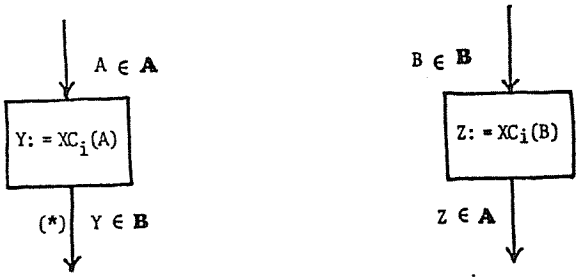


Figure 10. Parallel program flowchart fragments.

[6] This is somewhat artificial, because the procedural model on which axiomatic proofs are based is different from a functional model - in which the axioms are those of recursive function theory (see also IV.B). In this context let us assume that an exchange function is a name for a procedure which implements an exchange function.

Exchanges can be incorporated as follows. If there are flowchart-with-interpretation fragments as shown in Figure 10, and if these are the only two instances of exchanges in class i, then the interpretation shown is consistent. If there was an $XC_i(C)$, $C \in \mathbf{C}$ in a third process, then the strongest assertion possible on the arc marked (*), in the absence of further information, would be $Y \in \mathbf{B} \cup \mathbf{C}$, etc. This characterizes the information-passing capabilities of these primitives.

The other capability of these primitives is synchronization, incorporated by the appropriate constraint on execution sequences: the two operations in Figure 10 must be executed together, simultaneously (constraints such as these are discussed in [La3]). Thus termination of the left XC_i is proved by showing that the assertion "the locus of control of the left process is on arc (*)" eventually becomes true. The extension to XA's and XS's is straightforward.

C. Syntactic Analysis and Design Laws

The real strength of functional specifications, in the area of analysis, seems to be the use of design laws, syntactic analysis, and theorems to establish properties efficiently from the representation alone. Properties verified in this way are not arbitrary correctness conditions, but properties of general interest such as termination, freedom from deadlock, no loss of information, mutual exclusion, etc.

The reason for design laws is simply that worst cases are always intractable, and it is both prudent and respectable to refuse to deal with them. Investigating which cases can be analyzed efficiently shows us which design laws should be enforced.

The rest of this section will consist of preliminary results along these lines. They consist of three simple theorems concerning sufficient conditions for preventing non-termination of a process step because of pending, unmatchable XC's or XA's.[7] All depend on the highly important design law, or constraint, that the class subscript of an exchange is a constant rather than a variable. This makes it possible for static

[7] We will asume for simplicity that evaluations of all other primitive functions always terminate and that the successor functions are defined using control structures which do not allow infinite iteration or recursion.

analysis to find the interacting functions.

The first two theorems deal with analysis of particular sub-systems of the communication network, and the third one shows that these sub-systems can be analyzed independently. The choice of sub-systems follows a natural and frequently observed pattern.

Inter-process communication often uses XS/XA interactions in classes dedicated to that purpose (see Figure 4). In such cases, one could say that the processes executing XS's are producing resources needed by the processes with XA's to allow termination. As long as production of those resources does not halt, consumers will never be blocked for lack of them.

Theorem I: Let $P = \{p_1,p_2,\ldots,p_n\}$ be a set of processes using only exchange classes $\{C= c_1,c_2,\ldots,c_m\}$, and let no $p_i \in P$ evaluate either an XC_{c_j}, or both XS_{c_j} and XA_{c_j}, $c_j \in C$. Let $p_i \in P$ be described by (s_i,a_i), s_i, $a_i \in$ the power set of C, such that $c_k \in s_i$ iff. p_i evaluates an XS_{c_k}, and $c_k \in a_i$ iff. p_i evaluates an XA_{c_k}.

Let G be a directed graph whose node set is P, and such that there is an arc from p_i to p_j iff. $\exists\, c_k$ such that $c_k \in a_i$, $c_k \in s_j$. Thus the arc (p_i,p_j) is in G if p_i "consumes" XS's in a class "produced" by p_j. Then no XA in P will fail to terminate if:

(a) G is acyclic;

(b) unless an XA in p_i fails to terminate, p_i will eventually execute another XS in each class in s_i, $\forall p_i \in P$;

(c) $\exists p_i$ such that $c_j \in s_i$, $\forall\, c_j \in C$.

Proof: The hypothesis that no XA in p_i fails to terminate is proved by induction on the length ℓ_i of the longest path in G from p_i to a terminal node (which is well defined because G is acyclic), $\forall p_i \in P$.

The hypothesis is true if $\ell_i = 0$, because terminal nodes have no XA's.

If the hypothesis is true for all processes p_j such that $\ell_j = k$, it is true for all processes p_i such that $\ell_i = k+1$. p_i depends on processes with path length k or less to provide XS's to match its XA's, but there is at least one such process for each class, and it will always evaluate another XS in that class. p_i may be competing with other processes for these XS's, but the fair scheduling rule for exchanges states that it will not be locked out. □

The theorem is trivially applicable to the real time clock's interactions. It also applies to the message system of Figure 7. With n = 2, G is as shown in Figure 11.

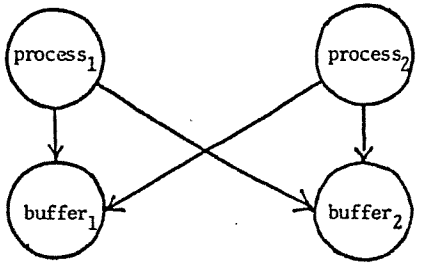


Figure 11. G for a two-process message system.

The second theorem concerns intra-process interactions (between asynchronous processes implementing the successor function), which often uses only XC/XC exchanges. The two possible purposes for such interactions at the functional specification level are (a) synchronization for reasons determined by the inter-process interface, and (b) transmission of information from the domain of one implemention function to that of the other. As an example of (b), in the processor-memory process mentioned in II.C, exchanges keep the state of the memory-implementing process completely separate from that of the CPU-implementing process, although both are part of the state of the implemented process. It is this factorization which makes a transformation to a set of completely asynchronous processes, with the memory process communicating with both processor and channel processes, almost trivial.

The following discussion of sufficient conditions for termination is sketchy and informal, but only for lack of space. Our first requirement is that the pattern of evaluation of XC's within a step of the implemented process be static and syntactically analyzable, so that it can be expressed as a precedence graph - in which the nodes are initiations or terminations of XC evaluations and the arcs are precedence constraints enforced by the structure of the functional specification.

A "path" through this graph is a traversal of its nodes which violates neither the precedence constraints nor the rules for matching of exchanges. The process step will always terminate if and only if all paths traverse all nodes. An example of a

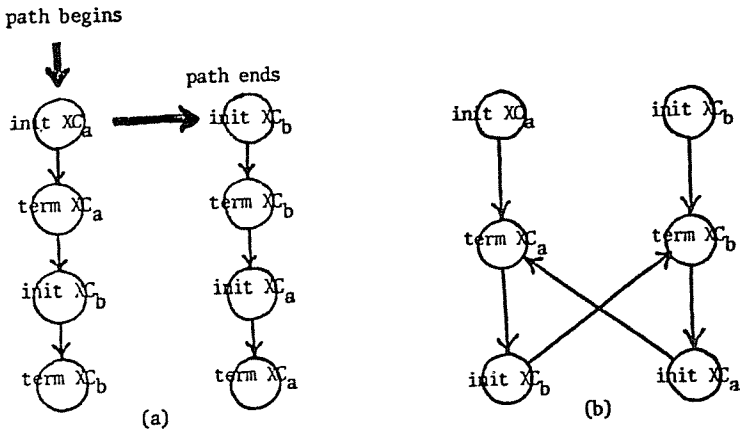


Figure 12. A non-terminating process step.

path which "gets stuck" before it can traverse all nodes is shown in Figure 12(a).

Unfortunately, this leads to an analysis algorithm which can be $O((2n)!)$, where n is the number of XC's. It is therefore impractical for large systems. But intra-process exchange patterns are often designed so that the matching is determined. A fixed matching pattern can be made trivial to detect (simply by the use of a different class for each matching XC pair), and can be analyzed for termination much more efficiently. In the graph for a process step with fixed matching, the two "terminate XC_i" nodes can be merged for all i, as shown in Figure 12(b). The graph then encodes all precedence constraints, including those induced by exchange matching. The process step will always terminate if and only if the graph is acyclic, an $O(\frac{3}{2}n)^2$ or better (depending on data structures) analysis algorithm.

Theorem II: Let p be a process which evaluates only a fixed pattern of XC exchanges in classes private to p, such that the matching between XC's is fixed by the specification of p's successor function. Let H be the graph with $\frac{3}{2}n$ nodes (n is the number of XC's evaluated), as described above, for p. Then no XC in p will fail to terminate if and only if H is acyclic.

Proof: H is a complete and accurate representation of all precedence constraints on a set of indivisible tasks, by definition. Any task set constrained by an acyclic precedence relation can be executed, and no task set constrained by a cyclic precedence relation can be. □

Finally, we must establish that these two types of analysis can be carried out independently.

Theorem III: Let $P = \{p_1, p_2, \ldots, p_n\}$ be a set of processes such that all exchange functions in P can be syntactically identified as participating exclusively either in inter-process interactions or in intra-process interactions. Let termination of all inter-process exchanges be verifiable using Theorem I, and termination of all intra-process exchanges be verifiable using Theorem II. Then no exchange in P ever fails to terminate.

Proof: No intra-system XC could fail to terminate because of the presence of inter-process XA's or XS's. Execution of XS's is transparent. Introduction of XA's in the H graph could not cause cycles because "termination of XA_i" has no predecessor in the graph except "initiation of XA_i."

No inter-process XA could fail to terminate because of the presence of intra-process XC's, as long as the XC's do not prevent termination of the process steps on which this XA is dependent, which is established by Theorem II. □

IV. CONCLUSIONS

A. Summary

Complexity is the major problem of systems design. It can only be solved through comprehensive models which make it possible to factor both the conceptual and analytic problems involved, to impose sufficient conditions for the solution of those problems, and to integrate th resulting solutions. Functional specifications of asynchronously interacting processes seem to be a step toward that goal. This is because they are general and compatible with any level of abstraction.

It has been shown how asynchronous interactions can be introduced into functional specifications by the exclusive use of three primitive functions, which also seem to be general and compatible with any level of abstraction.

Asynchronous interactions specified in this way are subject to analysis by both testing and axiomatic proof techniques. They are also subject to the definition of

sufficient conditions under which correctness properties are efficiently verifiable from syntactic analysis. It appears that some correctness properties may be more easily verified from functional specifications than from the more usual procedural models, because numerous data and control structure details need not be encoded.

B. Relation to Other Work

As mentioned previously, much work is now being done on high-level specifications of requirements and system designs. [Ro] and [HZ] offer introductions to SREM, ISDOS, HOS, and SADT, all of which are compared in [DV]. The formal representations of ISDOS and SADT differ from our notion of functional specification because they are not effective, i.e. subject to simulation.

The SREM representation does support simulation, but represents computational paths, for the purpose of requirement specification only, rather than digital process structures. We choose to represent digital process structures because (a) they allow the system design to be a refinement of the requirements, and (b) they are compatible with digital simulations of the real-world environments of proposed systems.

HOS representations are at a lower level of abstraction than our functional specifications, because all inter-module communication has already been bound to variables. The HOS axioms ensure that these shared variables retain their integrity and well-definedness throughout execution.

Work on axiomatic proofs and syntactic analysis of designs is abundant in the literature, of course. Much of it is based on an underlying procedural model; the axiomatic proof references are examples, as is [Ri] (in which a program model and an analyzable automaton are algorithmically related). Procedural models are intrinsically different from functional models in that the process structure which is explicit in the functional model is implicitly coded in the procedure/interpreter structure of the procedural model (this shows up in the variable, assignment, and scheduling concepts of the procedural model, all of which are absent in the functional model). Thus investigations of and operations on process structures are better done with functional models.

The work which is not based on an underlying procedural model, such as that on Petric nets, may prove useful for analysis of functional specifications.

C. Future Work

The research reported here is part of a larger effort to develop the concept of functional specification in theory and practice. The following subjects are among those being investigated: (a) a macro-based design language which will minimize the "parentheses-blindness" endemic to functional notations; (b) incorporation of resource and performance specifications; (c) further analytic tools and techniques; (d) a theory of equivalent transformations on process structures; and (e) a methodology for integrating functional specifications while satisfying the resource and performance requirements at all levels.

REFERENCES

[DV] Davis, Carl G., and Vick, Charles R. "The Software Development System." IEEE Transactions on Software Engineering SE-3, January 1977.

[FK] Frank, Howard, Kahn, Robert E., and Kleinrock, Leonard. "Computer.communication network design - Experience with theory and practice." AFIPS Conference Proceedings 40, 1972.

[HZ] Hamilton, Margaret, and Zeldin, Saydean. "Higher Order Software - A Methodology for Defining Software." IEEE Transactions on Software Engineering SE-2, March 1976.

[Ke] Keller, Robert M. "Formal Verification of Parallel Programs." CACM 19, July 1976.

[La1] Lamport, Leslie. "Proving the Correctness of Multiprocess Programs." To appear in IEEE Transactions on Software Engineering.

[La2] Lamport, Leslie. "Time, Clocks, and the Ordering of Events in a Distributed System." Massachusetts Computer Associates, Inc., 1976.

[La3] Lamport, Leslie. "Towards a Theory of Correctness for Multi-user Data Base Systems." Massachusetts Computer Associates, Inc., 1976.

[OG] Owicki, Susan, and Gries, David. "Verifying Properties of Parallel Programs: An Axiomatic Approach." CACM 19, May 1976.

[Ri] Riddle, William E. "An Approach to Software System Modelling, Behavior Specification and Analysis." To appear in CACM.

[Ro] Ross, Douglas T.,ed. Special Collection on Requirement Analysis. IEEE Transactions on Software Engineering SE-3, January 1977.

Appendix B Petri Net Models

B.1 Petri Net Abstractions of Functional Specifications

In this section we will describe interactions of asynchronous processes formally in terms of Petri nets. These asynchronous processes are assumed to be defined in the functional notation developed in [Fi77], and in particular our discussion will center around the primitive exchange functions which logically determine these interactions (see Appendix A of the preceding reference). However, our models will ignore state values entirely and so will model only the major steps in the evaluation of the state successor function. In short, we will have only a rudimentary flow of control abstraction of state successor function evaluation. It is important to note that since a loss of (state value) information is involved in the construction of our models that it is impossible to perform an inverse transformation on the models in order to recover the original functional specifications from which they are derived. Further, we do not propose that our models serve as a medium for system design or elaboration; on the contrary they are to be considered as components of automated system analysis tools which will remain transparent to the user. Our main purpose then is to describe a formal method for representing stages

---

[Fi77] Fitzwater, D. R., "The Formal Design and Analysis of Distributed Data-Processing Systems," University of Wisconsin Computer Sciences Department, Report CSTR 295, March 1977.

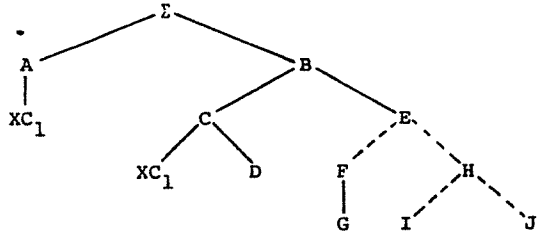
in the evaluation of state successor functions. Use of the Petri net models is exemplified in the detection and prevention of logical blockage or in calculation of minimum and maximum time bounds for the evaluation of functions. We must realize, however, that because of the complexity of the calculations involved in the complete characterization of our Petri net models this approach is limited to smaller numbers of process interactions and that it is merely one design tool to be used in conjunction with others, such as the conditions for allowed specifications of Appendix B in [Fi77].

We have relegated the definitions of Petri net properties and the formal detailed construction of our models to the Appendix. In the remainder of this section we will present two examples of model construction which will provide an intuitive appreciation for the implications of the rules of model construction. Our first example models a single system with state successor function Σ (not to be confused with the notation for a state space in Appendix A of [Fi77]) with two component successor functions A and B. In a linear, somewhat symbolic notation we write Σ as follows:

$$\Sigma(\sigma_A,\sigma_B) = (A(XC_1(U)),B(C(XC_1(V),D(W)),$$
$$E(P:F(G(X));H(Q:I(Y);(J(Z)))))).$$

Here P and Q are predicates and we have separated the argu-

ments of selector functions by semicolons rather than commas for the sake of clarity. The precedence graph for Σ is as follows, where dotted lines indicate selector function arguments:



A diagram representing the Petri net (but without the initial marking of a single token in the place for Σ) is given in Figure B.1-1.

The only unusual feature about our figure is that places have been represented as circles, squares, and squares superimposed upon circles. This is purely a metanotation for the eye and does not imply any difference in behavior between the variously shaped places. It simply indicates in those cases where a function is represented in two places, for example E, that the presence of a token in the circular place denotes the initiation of the evaluation of that function, and that the presence of a token in the square place denotes the completion of the evaluation of that function. In the Appendix

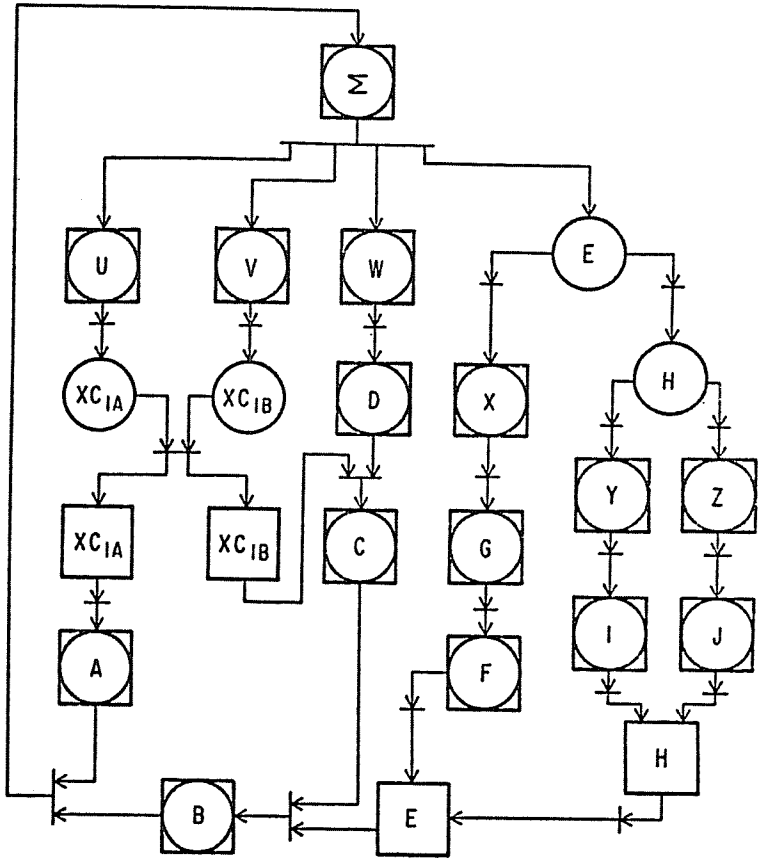
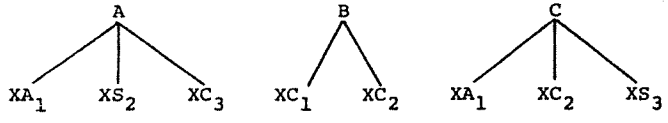


Figure B.1-1. Petri Net Model of a Single Process

we refer to these two nodes, for example in the case of E, as entry-E and exit-E respectively. Where this distinction is not made, for example as with X, a circle with a square superimposed upon it is the sole place for representing the evaluation of the function associated with it. Comparison of the Petri net model and the precedence graph for the system reveals a close structural correspondence. Further, it is easy to visualize how reachable markings of the net can model possible orders of evaluation of Σ. For example, the completion of the evaluation of the selector function H is modelled by a firing in which a token from the place for I or the place for J is passed to the square place for H (i.e., exit-H).

Our next Petri net construction models the interactions of three asynchronous processes whose precedence graphs are given below:



This simple example is presented primarily for the illustration of the modelling of exchange interactions rather than the modelling of multiple levels of nesting of functions,

including selector functions, as in the first example. The initial marking of the graph in Figure B.1-2 has by definition a single token in each of the three topmost places but in no others. Certainly the construction of this graph should be studied by following carefully the rules in the Appendix if an understanding of their use is to be achieved. (Note that both examples above jointly use every rule of construction in the Appendix at least once.)

Several features of exchange interaction are illustrated in the figure. For example, note that there are pairs of arcs leading to transitions from the pair of places XA_{1A} and XC_{1B} and from the pair XC_{1B} and XA_{1C} but not the pair XA_{1A} and XA_{1C} (all entry places) for the simple reason that XA functions in the same class are not allowed to exchange with one another. (Note that the second subscript of the place name merely identifies different occurrences of exchange functions in the same class.) Somewhat less obvious is the role of the latch places (represented as ovals in Figure B.1-2) in exchange classes which contain XS functions (see section B.2), for example as in class 3 above. When we note that an entry place and its corresponding latch place never contain a token simultaneously then the purpose of the latch place becomes clearer. In Figure B.1-2 we see that, for example, if latch-XS_{3C} contains a token then it can produce a firing only if entry-XC_{3A} also contains a token (i.e., XC_3 is pending in process A) or

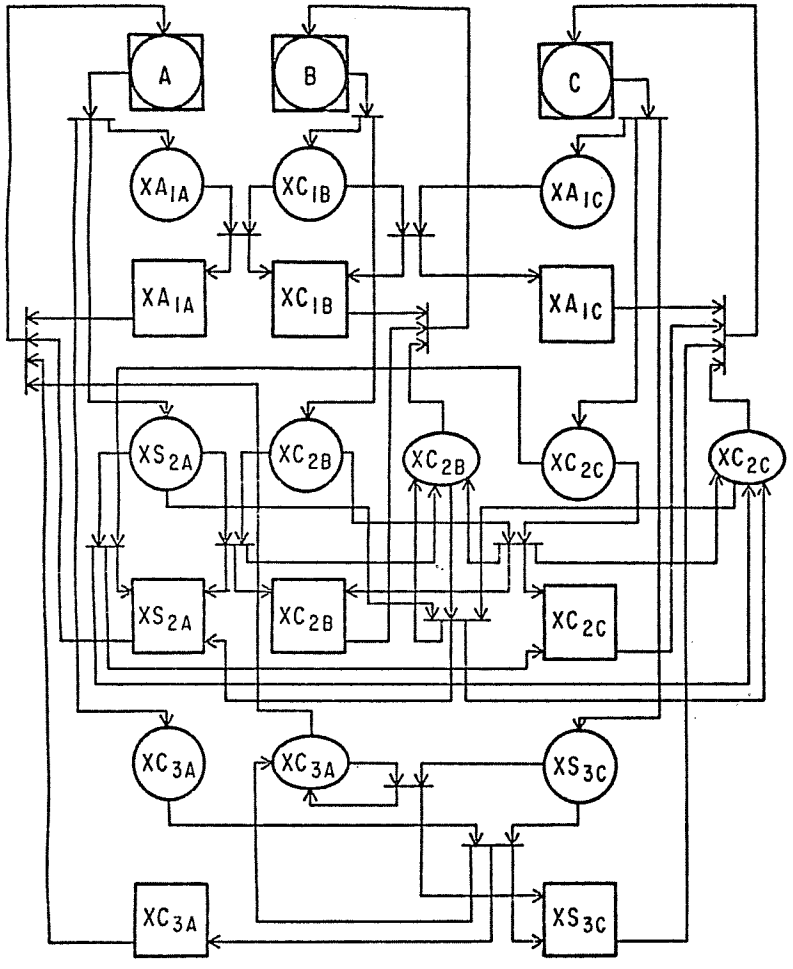


Figure B.1-2. Petri Net Model for Three Asynchronous Processes A, B, and C.

if latch-XC_{3A} also contains a token (i.e., XC_3 has already exchanged in the current system step of process A and so is not pending).

Although we provide a formal construction for our Petri net models of asynchronous interaction, we have said nothing about the consistency of the construction or the correctness of the model in actually simulating the interactions which are delimited by functional specifications. Proofs for both the consistency and correctness at this point would probably not be very illuminating in showing either how Petri net simulations evolve or why the particular construction used here is an adequate model for the gross (i.e., state space independent) features of state successor function evaluation. These proofs involve a step-by-step analysis of the syntax of functional specifications along with the rules of Petri net construction given later and so are lengthy and tedious. Another property of our Petri net models which requires a similarly lengthy proof is boundedness. In fact, we can assert that any place in our Petri net models may contain no more than one token in any reachable marking. This limits the number of reachable markings to 2^n for a Petri net with n places although the actual number of markings may be well below this upper bound.

B.2 Formal Construction of Petri Net Models

In this section we will first define Petri nets and their properties and then give a method for constructing Petri net models from functional specifications. The definitions which follow are taken directly from [LaR75].

A Petri net is a quadruple.

$$P = \langle P,T,A,M_0 \rangle$$

where P is a finite set of places; T is a finite set of transitions or firing bars; A is a finite set of arcs, $A \subseteq (P \times T) \cup (T \times P)$; and M_0: $P \to N$, N the set of natural numbers, is the initial marking.

Initially each place p of the Petri net contains $M_0(p)$ tokens. Let t be a transition. Then $\{p \mid (p,t) \in A\}$ and $\{p \mid (t,p) \in A\}$ are called the input places (inputs) and output places (outputs) respectively of t.

Transition t is enabled or fireable when each input place of t contains at least one token. If t is enabled, then it may be fired which results in the removal of one token from each input place of t and the addition of one token to each output place of t. If t is not enabled, then it is disabled. Write

[LaR75] Landweber, L. H. and E. L. Robertson, "Properties of Conflict Free and Persistent Petri Nets," Computer Sciences Technical Report #264, University of Wisconsin, December 1975.

$M_1 \xrightarrow{t} M_2$ ($M_1 \xrightarrow{t}$) to indicate that t is enabled by the marking M_1 and that the firing of t yields marking M_2 (t is enabled by the marking M_1). Extend the notation and definitions to sequences of transitions, $\sigma \in T^*$, called firing sequences.

The set of reachable markings or the reachability set R_P of the Petri net $P = \langle P,T,A,M_0 \rangle$ is $\{M \mid M_0 \xrightarrow{\sigma} M$, for some $\sigma \in T^*$. If $M \in R_P$ we say that M is reachable in P. The reachability problem for a class C of Petri nets is the problem of deciding, given an arbitrary $P \in C$ and marking M, whether $M \in R_P$.

A place in a Petri net is bounded if there is a $c \in N$ such that for all reachable markings M, $M(p) \leq c$. A Petri net is bounded if each place in the net is bounded.

Our construction of Petri net models for evaluation of functional specifications will be described below by a series of detailed but straightforward rules. (Close examination of the examples in section B.1 will be essential to an understanding of the use of these rules and their purpose. In a slight departure from conventional practice our diagrams represent the components of the Petri nets as follows: a place is a circle, square, ellipse, or a square superimposed upon a circle, a transition is a bar, and an arc consists of one or more connected line segments, where as in block wiring diagrams the crossing of line segments implies no connection.) We recall from [Fi77]

[Fi77] Fitzwater, D. R., "The Formal Design and Analysis of Distributed Data-Processing Systems," University of Wisconsin Computer Sciences Department, Report CSTR 295, March 1977.

that a functional specification for a system consists of a state successor function which is composed of one or more component successor functions, where the completion of a process step corresponds to the evaluation of these (possibly non-deterministic) functions. We may deal with collections of one or more such systems and we describe asynchronous interactions within a single system as intraprocess exchanges and interactions between systems as interprocess exchanges as in [Fi77].

One bit of new notation is introduced at this point in order to distinguish several different occurrences of a function name in a set of specifications. We simply append a unique, and possibly additional, subscript to each of those occurrences of the function name (as with XC_{1A} and XC_{1B}, which in the second example (B.1) are two occurrences of the function XC_1 in systems evaluating functions A and B respectively). However, it should be emphasized that this extra subscript is not a part of the functional specification and is purely a device for illustration of the construction of Petri net models.

Assuming that we have functional specifications for a collection of systems, we can now derive the components of a Petri net model for the evaluation of these systems according to the following lists of rules. First we derive places for the Petri net.

(1) Every function f which is not an exchange function or selector function has a single place representing it. The

presence of a token in such a place indicates either that the function is being evaluated during the current system step or that the use of results of that evaluation is pending, contingent upon some other system event (i.e., firing), during the current system step. We say that this place is both the entry place and the exit place for the function f (abbreviated as entry-f and exit-f, respectively), and we represent it graphically as a square superimposed upon a circle.

(2) Every simple argument x of a function f (where we define a simple argument as any argument which is not itself a function) has a place which we describe as both entry-x and exit-x as in (1) above. (This rule is used in the first example in section B.1.) When a token is present in such a place it is taken to mean that the argument is available for use in the evaluation of the function with which it appears. However this rule is not used if we wish to model exchange functions with simple arguments such that process interaction is pending without delay as soon as the argument is available, for example, at the beginning of a system step. (This alternate choice of modeling depends on the particular implementation of state successor function evaluation being employed, which is beyond the scope of the present discussion.) In the latter case we omit any places for simple arguments of exchange functions.

This omission of places occurs in the second example of section B.1 and will be discussed in succeeding rules wherever relevant.

(3) Every exchange function has a corresponding entry place and in addition a distinct exit place. (Recall that we refer to the ith occurrence of XC_n, for example, as XC_{ni}, and we will by extension label the Petri net places for this occurrence of XC_n as entry-XC_{ni} and exit-XC_{ni}.) Furthermore, every occurrence of an XC or XA function which is in an exchange class containing an XS exchange has a third corresponding place called a latch place, for example, latch-XC_{ni} if exchange class n contains an XS_n. A token in an entry place for an exchange function denotes pending evaluation of that instance of the function in the current system step. On the other hand, a token in the exit place of an exchange function corresponds to the scheduling of the particular occurrence of that function for an exchange of arguments, that is, pairing with another occurrence of the function in the same class (in the case of XC's or XA's) or possibly self-exchange in the case of an XS function for which no XC or XA in the same class is pending at the time of evaluation. (The latch places are in fact present in order to prevent an immediate exchange from performing self-exchange if, as we have said, an XC or XA in the same class is pending as illustrated in

the second example in section B.1.) We have represented entry, exit, and latch places consistently as circles, squares, and ellipses respectively in the diagrams of section B.1.

(4) Every selector function g has an entry place, entry-g, and a distinct exit place, exit-g. A token in the entry place corresponds to the initialization of evaluation of the function, and a token in the exit node corresponds to the completion of the evaluation of one of the arguments of the selector function. This construction is illustrated in the first example in section B.1.

(5) Corresponding to every state successor function f there is a single place which is both entry-f and exit-f. A token in this place indicates that all calculations for a system step have been performed and that a new system step is ready to begin.

Before we list the transitions and arcs of our Petri net model in terms of the places already given above we define initiation places of functions and arguments recursively as follows:

(1) The only initiation place for a simple argument is its (combined) entry and exit place.

(2) The only initiation place of a selector function is its entry place.

(3) The set of initiation places of a function other than a selector function consists of precisely:

(a) those places which are the entry places (and hence exit places) of its simple arguments, and

(b) those places which are the initiation places of all arguments which are themselves functions (i.e., not simple arguments).

The motivation for the definition of initiation places just given becomes clear from the construction rules and examples of section B.1. (Again a complication arises if we omit places for simple arguments of exchange functions as discussed in rule 2 for Petri net places above. We could modify our definition of initiation places above in order to reflect the latter alternative, since obviously in that case a simple argument of an exchange function would have no place representing it. However, we can avoid this difficulty and others by constructing the Petri net according to the original definitions and then by transforming the complete net by a simple rule given later.)

Transitions and arcs for our Petri net model are defined by the following rules:

(1) For every function f not a selector function there is a transition p. In addition, there exist arcs from the exit place of every argument of the function to p, and also there exists an arc from p to the entry node of f.

(2) For each argument of a selector function g:

(a) there is a distinct transition p and an arc from the exit place of the argument to p and an arc from p to the exit place of g, and

(b) there is also a distinct transition q with an arc from the entry place of g to q and an arc from q to the initiation places of the argument.

(3) For every exchange class n and every pair of instances of exchange functions in that class, XC_{ni} and XC_{nj} (or XA_{nj}) there is a transition p which has arcs from entry-XC_{ni} and entry-XC_{nj} (or entry-XA_{nj}) to p and which also has arcs from p to exit-XC_{ni} and to exit-XC_{nj} (or exit-XA_{nj}). In addition if exchange class n contains immediate exchanges then there are arcs from p to latch-XC_{ni} and to latch-XC_{nj} (or latch-XA_{nj}).

(4) For every exchange class n and every pair of instances of exchange functions in that class, XS_{ni} and XC_{nj} (or XA_{nj}), there is a transition p which has arcs from entry-XS_{ni} and entry-XC_{nj} (or entry-XA_{nj}) to p and which also has arcs from p to exit-XS_{ni} and to exit-XC_{nj} (or exit-XA_{nj}). In addition, there is an arc from p to latch-XC_{nj} (or latch-XA_{nj}).

(5) For every exchange class n containing immediate exchanges and every instance of an exchange function in that class, XS_{ni}, there is a transition p which has arcs from entry-XS_{ni}

and all latch-XC_{nj} places and all latch-XA_{nk} places to p. Furthermore, there are transitions from p to exit-XS_{ni} and all latch-XC_{nj} places and all latch-XA_{nk} places.

(6) There is a transition p for each system with arcs from the exit node of every component successor function to p. In addition, there are arcs from p to the initiation places of every component successor function in a system. A firing involving p corresponds to the completion of a step.

In order to complete our derivation of the Petri net model we must specify an initial marking. We simply place one token in the place for every state successor function and no tokens elsewhere. This marking corresponds to the beginning of a system step in each process. We conclude our construction of the Petri net model by giving the optional transformation which eliminates places for simple arguments of exchange functions. This transformation has been anticipated several times in the text and is quite simple. By the original construction we have for each simple argument of an exchange the following: (1) an arc from some transition t to the place for that argument, (2) an arc from that place to (3) a transition u unique to that argument, and finally (4) an arc from u to some place p. We replace (1)-(4) above by an arc from t to p in each case, thus completing the construction. This transformation is always well-defined although we shall not give a proof of that

fact. The result of this transformation was illustrated in the second example of section B.1.

Appendix C

Formal Syntax Definitions

Here we will present the notation for syntactic definitions. The notation is based on N. Wirth's suggestions ["What Can We Do About the Unnecessary Diversity of Notation for Syntactic Definitions," CACM, November 1977]. The only exception to Wirth's notation are that we (1) distinguish between left (`) and right (´) quotation marks, and (2) write a left (or right) quote singly when it appears contained within quotes, not doubly as does Wirth. The following discussion closely follows Wirth.

We first define the metalanguage using the metalanguage, and then provide a brief explanation.

```
    SYNTAX = {PRODUCTION}.
PRODUCTION = IDENTIFIER`=´EXPRESSION`.´.
EXPRESSION = TERM {`|´TERM}.
      TERM = FACTOR{FACTOR}.
    FACTOR = IDENTIFIER|LITERAL|`(´EXPRESSION`)´|
                 `[´EXPRESSION`]´|`{´EXPRESSION`}´.
   LITERAL = ``´CHARACTER{CHARACTER}`´´.
```

{ } denotes zero or more occurrences of the enclosed expression. It resembles the Kleene star, as for example, {a} is a^* which is ε|a|aa|aaa.... Note that ε denotes the empty string.

[ ] represents zero or one occurrence of the enclosed expression. For example, [a] is ε|a.

Left and right parentheses are used for grouping purposes. For example, (a|b)m is am|bm.

Terminal symbols are enclosed within left and right quotes. For example, `{´ is the terminal symbol {, not the metasymbol {.

The nonterminals in the above productions are SYNTAX, PRODUCTION, EXPRESSION, TERM, FACTOR, LITERAL, IDENTIFIER, and CHARACTER. IDENTIFIER denotes a nonterminal, LITERAL denotes a terminal, and CHARACTER denotes one of the characters in the particular character set chosen. For brevity and lack of a fixed character set, IDENTIFIER and CHARACTER are not defined further.

A production then is a left side (always a nonterminal in our formal grammar), the "=" symbol and a right side (consisting of terminals, nonterminals, and metaoperators). For example:

```
LETTER_A = CAP_A|SMALL_A
```

reads: LETTER_A produces CAP_A or SMALL_A

What we mean by this is that the nonterminal element "LETTER_A" can be written as either the nonterminal "CAP_A" or the nonterminal "SMALL_A". In this formal way, we may go on to specify "LETTER_A" as:

```
LETTER_A = CAP_A|SMALL_A
SMALL_A = `a´.
CAP_A = `A´.
```

Both `a´ and `A´ are terminals in the grammar.

If these three productions were to specify an entire grammar, then the language would be the set {A,a}.

Modifying the grammar somewhat we get the following set of productions:

```
LETTER_A = CAP_A|SMALL_A
SMALL_A = `a´{`a´}.
CAP_A = `A´{`A´}.
```

This grammar will produce an infinite number of sentences. Examples of such sentences are:

a

aa

aaa

A

AA

AAAAA

etc.

The use of Wirth's notation allows a compact and clear representation for a context-free language.

Appendix D - An Investigation of Digital System Equivalence in the Context of a Comprehensive Design Theory

This appendix consists of the original unedited text of a paper by Pamela Zave. The original page numbering, to which the table of contents of this paper refers, can be found in the upper right hand corner of each page. Page numbering for the present report continues uninterrupted at the bottom center of each page.

Technical Report TR-592 November 1977

AN INVESTIGATION OF DIGITAL SYSTEM EQUIVALENCE IN THE CONTEXT OF A COMPREHENSIVE DESIGN THEORY*

by

Pamela Zave

* This work was supported by the Scientific Services Program, Battelle Columbus Laboratories, under Contract DAAG29-76-D-0100.

Abstract

In the context of the comprehensive design theory being developed by Fitzwater, Zave, et. al., the problem of digital system equivalence takes on a new and more tractable form. The design steps in which equivalence is useful are identified. For these steps, a set of equivalence preserving transformations is identified, proved to be equivalence-preserving, and shown to be useful by examples. This approach to equivalence is compared to other work, with reflections on the design theory.

TABLE OF CONTENTS

I. INTRODUCTION 1

II. THE EQUIVALENCE PROBLEM 3

A. Definition 3

B. Equivalence of Open Systems 4

C. Equivalence in Closed Systems 7

III. AN OPTIMIZATION-ORIENTED VIEW OF SPECIFICATIONS 9

A. An "Evaluation" Specification Language 9

B. Relevant Analyses 12

C. A Particular Definition of Equivalence 19

D. Preliminary Observations 21

IV. A FUNDAMENTAL THEOREM 23

V. EQUIVALENCE-PRESERVING, OPTIMIZING TRANSFORMATIONS 26

A. A Classification 26

B. Transformations of Type 1a: "Process Splitting" 27

C. Transformations of Type 1b: "Scheduling" 31

D. Transformations of Type 2a: "Resource Sharing" 35

E. Transformations of Type 2b: "Delay Elements" 37

VI. COMPARISONS TO OTHER EQUIVALENCE STUDIES 39

VII. CONCLUSIONS 40

REFERENCES 43

AN INVESTIGATION OF DIGITAL SYSTEM EQUIVALENCE IN THE CONTEXT OF A COMPREHENSIVE DESIGN THEORY

I. INTRODUCTION

This is the final report on work done under Scientific Services Program Contract DAAG29-76-D-0100, Battelle Columbus Laboratories.

The scope of this contract was to investigate the equivalence problem in the context of the digital system design theory being developed by D. R. Fitzwater at the University of Wisconsin - Madison, the author, and others, the point being to determine whether equivalence-preserving transformations were useful and feasible in the system development process. For information about the design theory, the reader is referred to [Fi], [ZF], and [FZ] in the published literature, interim reports by D. R. Fitzwater on BMDSC-ATC-P Contract DASG60-76-C-0080, and finally to the reports still in preparation on this work.

The goal of the design theory is to produce a representation, method, and decision theory for the development of digital systems. The representation is a formal system specification language which ensures that relevant properties of systems are efficiently decidable. The method is a heuristic effective procedure, i.e. an automated procedure operating on the representation and using the interactive guidance of a human designer, which develops the system by successive transformations. The decision theory contains the information and tools which the human designer uses to guide the procedure toward the most efficient development and the best developed system.

Part II of this report discusses the role of equivalence in this design theory. Parts III and IV present the basic theoretical approach to the form of equivalence found most useful. In Part V, specific equivalence-

preserving transformations, equivalence proofs, and examples are given. Parts VI and VII compare this work to other approaches to equivalence, and present conclusions and recommendations.

II. THE EQUIVALENCE PROBLEM

A. Definition

In its most general form, the equivalence problem is to determine whether or not two arbitrary formal objects are identical with respect to a set of formal attributes. The set of attributes defines what is important; the objects are equivalent only if they are alike (equal-valued) in important characteristics (attributes), but may differ in other characteristics . Thus equivalence is the natural vehicle for expressing formally the invariance of properties, and can be used regardless of the level of abstraction at which the properties are defined.

The equivalence problem is also very hard, and most non-trivial cases are undecidable many times over. For this reason we must confine ourselves to studying the equivalence of very similar objects. Fortunately, this is consistent with our need: to show that certain properties are preserved under simple changes to formal system specifications.

At present, only "logical" or "functional" attributes are sufficiently well understood to have been elaborated in our specification language. Other attributes, such as comments and "performance" attributes, are now represented only as undifferentiated "attribute lists" associated with certain syntactic entities. Thus we can only deal explicitly with logical attributes here, while ensuring that all results will be applicable or extensible to later versions of the specification language.

Since useful applications of equivalence theory are to be defined by steps of a design process (i.e. simple changes in a specification), we will next consider what these steps might be.

B. Equivalence of Open Systems

Two common design steps are partition and assembly. In a partition step, a system is broken into subsystems, each to be carried independently through further design steps. An assembly step is used to put the subsystems back together later. The subsystems are "open systems," because they exist for their interactions with other subsystems, and are not self-contained.

Partition and assembly steps themselves do not involve any interesting equivalence problems, but they do require that some strong form of behavioral ("black-box") equivalence be preserved, for all subsystems, during the steps between partition and assembly. Otherwise a subsystem could not be operated on independently, and partitioning would be vacuous. We will now show that this requirement makes it impossible to carry out any kind of usefully top-down, independent design of the subsystems. Since this follows directly from the definitions of partition and assembly steps, the conclusion is that these steps have no place in a top-down design process.

This form of design might work[1] if it were possible to specify the behavior of a subsystem independent of its internal structure, and to relate the structure-independent description to the various structures it might have. The former is necessary for partitioning without overly constraining the subsystems; the latter is necessary for proving that the behavioral interface is preserved.

Neither of these is impossible if the subsystem is merely a sequential procedure: the interface description consists of an argument set, a value set,

[1] There is the additional problem of allocation of performance attributes, which has no apparent solution in this form, but is outside the scope of this report.

and an assertion relating the two; this description can be related to procedure code using axiomatic proof techniques (see [Ho] for a seminal reference).[2]

In the interesting case, however, the subsystem is itself a set of asynchronously interacting parallel processes. Behavioral descriptions of these have been investigated in [Za] and [Ri]. Consider, for example, the subsystem of Figure 1, in which the action of process A is to emit a message "a" on each step, and the action of process B is to emit a "b" on each step. Its behavioral interface must be described in terms of the sequences of messages received by the environment, since messages are the only form of interaction here. The behavior could be described as {a, b, aa, ab, ba, bb, aaa, aab, aba, abb, baa, bab, bba, bbb, aaaa, aaab, aaba, aabb, abaa, abab, abba, abbb, baaa, ...} , which is an infinite set of sequences suffering from combinatorial explosion due to the arbitrarily varying relative rates of the two processes. Such a description has the virtue of

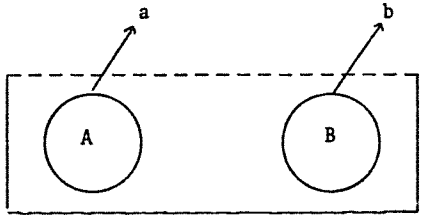


Figure 1. An open subsystem

[2] Note that this cannot be done in all cases, but "structured programming" rules are intended to weed out the intractable ones.

being structure-independent, but it is difficult to imagine any means by which it could be related to particular structures, either to verify or to generate them.

Using the event expression notation of Riddle ([Ri]), on the other hand, the behavior can be described as $a^* \Delta b^*$ - where the Δ ("shuffle") operator denotes an arbitrary interleaving of strings from a^* and b^*. This expression is simple because it reflects the process structure. It is not structure-independent, and it cannot be used to verify or generate structures other than the one from which it came, because equivalence of event expressions is undecidable!

We conclude that there is no way to describe a behavioral interface that is both independent of and commensurate with parallel process structure, and that in the absence of such a scheme, we must deal only with closed systems (Figure 2).

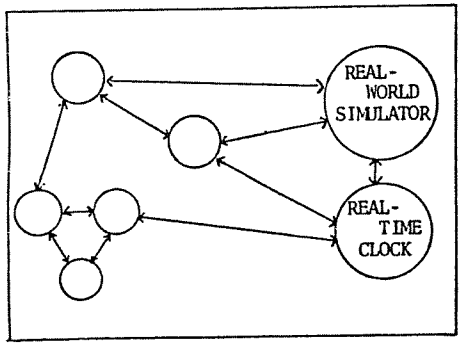


Figure 2. A closed system.

C. Equivalence in Closed Systems

The possible design steps on closed system specifications are elaboration, evolution, optimization, decomposition, and integration. Decomposition refers to breaking a specification into several complete specifications of the same system, each with respect to a different set of attributes. Integration refers to the recombination of these, after independent design steps. These are interesting design operations, but are not sufficiently well understood at this time to be dealt with here.

Elaboration is an operation which adds additional attributes by replacing primitive entities by non-primitive entities, and is the fundamental design step. Sufficient conditions must be defined for elaboration steps so that if they begin with valid, complete and consistent specifications, they end with valid, complete, and consistent specifications having the same attributes as before, and also some additional attributes. No additional equivalence concepts appear to be needed.

Evolution is an operation which involves backtracking to a less elaborated version and re-elaborating. It also appears to need no additional equivalence concepts.

The final type of step, optimization, transforms a specification into a more desirable form. Specifically: an elaboration step defines a (formerly primitive) entity, an evolution step redefines an entity in such a way that the old and new definitions need not be related, and an optimization step redefines an entity in such a way that the old and new definitions are logically, or functionally, equivalent. Obviously, here equivalence assumes its importance in the design method.

How might the equivalent definition be more desirable? There seem to be two ways. At some point the formal development process stops, and its present state is realized as a digital system. A more desirable form may be

one which is closer to the form of the intended realization, and thus takes less intellectual effort to realize.

The other possibility involves performance attributes, which are not yet elaborated. But consider this hypothetical (yet probable) example: Every function declared in a specification has an attribute which is an estimated evaluation time. This attribute of primitive functions is supplied by the designer; for non-primitives it is calculated from the attributes of primitive functions used in its defining expression. Now if a non-primitive function is defined in terms of a sequence of four primitive function evaluations, each estimated to take n seconds, the time attribute of this function will be 4n . If an optimizing transformation could find a logically equivalent expression in which there were two parallel evaluations of two functions each, it could also recalculate the time attribute to be 2n. This corresponds to the traditional idea of opitmization.

Based on this reasoning, the rest of this report will concern itself with optimizing transformations on closed systems.

III. AN OPTIMIZATION-ORIENTED VIEW OF SPECIFICATIONS

A. An "Evaluation" Specification Language

Optimizations will be operations on specifications as data objects. Unfortunately, there are two quite different principles which can be used to organize the data:

(1) the definitional principle, under which the specification is a block-structured set of definitions;

(2) the evaluation principle, under which a specification is a graph structure isomorphic to the function evaluations occurring when it is interpreted.

The specification language being designed as part of our theory follows, for sound reasons, principle (1). The problem is that equivalence concerns evaluation structures, and thus optimizations are most conveniently carried out on data objects organized by principle (2).

The ultimate solution to this problem is a matter of good design of a database and algorithms on it. It would be premature to attempt it here. Our interim solution is to define an "evaluation" language (based on (2)) and use it for present purposes. A grammar for this language is given in Figure 3.

The correspondence between the definition and evaluation languages should be intuitive. These are the most important ideas:

(1) In the former, a non-primitive set or function can be defined as an expression in terms of other sets or functions, and that definition can be used in many places. In the latter this is not possible: only primitive set and function names appear, and structure must be explicitly specified everywhere it is used.

(1) $\forall\, k \in N$:

$< \text{sys-spec} > \rightarrow ((< \Sigma_1 >, < f_1 >), \ldots, (< \Sigma_k >, < f_k >))$

(2) $\forall\, k, \ell \in N$:

$(< \Sigma_k >, < f_k >) \rightarrow (\Sigma_{k1} \times \cdots \times \Sigma_{k\ell}, (< f_{k1} >, \ldots, < f_{k\ell} >))$

(3) $\forall_i \in N^+$, $k \in N$, $c \in N$, $s \in N$:

$< f_i > \rightarrow (< f_{i1} >, \ldots, < f_{ik} >)$

$\rightarrow f_i\ (< arg_{i1} >, \ldots, < arg_{ik} >)$

$\rightarrow [< f_{i1} > : < f_{i2} >, \ldots, < f_{i,2k-3} > : < f_{i,2k-2} >,$ true : $< f_{i,2k} >]$

$\rightarrow XC^s_c\ (< arg_{i1} >)$

$\rightarrow XA^s_c\ (< arg_{i1} >)$

$\rightarrow XS^s_c\ (< arg_{i1} >)$

(4) $\forall_i \in N^+$, $k \in N$, $\ell \in N$:

$< arg_i > \rightarrow < f_i >$

$\rightarrow \sigma_{k\ell}$

f_i

Figure 3. Grammar of the evaluation language. N is the set of positive integers, and N^+ is the set of finite sequences of positive integers. < > indicates a non-terminal.

(2) In the former, a primitive set or function can be given any name, and used in many places. In the latter, each instance is given a unique name based on its place in the structure.

A formal definition of this correspondence shall be relegated to the design of the language interpreter and database. As the evaluation language is not necessarily suited for any analyses but those of immediate interest, we will assume that all specifications are complete and consistent in their definitional form.

In the evaluation language the notion that some values yielded by exchange evaluations are needed in several places must be encoded explicitly, because the language requires a separate exchange instance in each place. Superscripts do this job: if several instances in the same process, with the same type and class, have the same superscript, they represent only one evaluation. Otherwise they represent several evaluations.[3]

Non-terminal names in the grammar can and will be used to refer to the substructures they generate. Not all structures which could arise in a definitional specification can be named, however,

We refer specifically to the fact that the "top level" of the specification is a simple cross-product of sets and a corresponding simple tuple of functions. This includes forms which are isomorphic to a tuple of tuples, a single successor function, etc. What is leaves out is the case where a single function yields a structured value for the next state. We are ruling out the direct expression of that case for the purpose of having a clean notation relating state components and the functions which compute their values.

[3] Since only exchange functions have side effects, they are the only functions for which single or multiple evaluation makes a logical difference.

The importance of this should not be exaggerated. The evaluation grammar serves to name things so that they can be manipulated easily. In the cast at hand we are merely refusing to name, not precluding in the definition language, a structure which has no particularly useful manipulations.

B. Relevant Analyses

In this section two simple analysis algorithms, needed for subsequent discussions, will be presented. They operate on specifications generated by the grammar in Figure 3.

The first algorithm forms from a process specification (generated from $(< \Sigma_j >, < f_j >)$ in the grammar) a finite set of exchange precedence graphs, any one of which may characterize the exchange behavior of a step of the process. There must be a set of grpahs because each selection construction may create a value-dependent set of different possibilities for what can happen.

The grammar attaches structural names to both non-terminals and primitive functions. In this presentation the structural name of an exchange function will be stored in the graph as a pointer to the location of the exchange in the specification.

The nodes of the precedence graph are triples (type, class, pointerset), where type $\in$ {XC, XA, XS}, class $\in N$, pointerset $\in 2^{N^+}$. Let G be the set of all precedence graphs with nodes of this type.

We will also generate preliminary graphs whose nodes are quadruples (type, class, pointer, superscript), type $\in$ {XC, XA, XS}, class $\in N$, pointer $\in N^+$, superscript $\in N$. Let G^P be the set of all precedence graphs with nodes of this type or distinguished null nodes.

Let A be the set of all constructions which can be generated by the specification grammar from $< arg_i >$, $i \in N^+$. Then computation of the precedence graphs requires two functions, graph : $A \to 2^{G^P}$ and connect : $G^P \to G$, defined below.

The "graph" function maps a specification fragment onto a set of preliminary precedence graphs. It is defined recursively, and graphs are represented pictorially. A circle is a node and a box is a graph or subgraph. It is to be understood that if an application of "graph" to a partially completed member of a result set yields a set value, then the elements of this new set are all added individually to the result set - not as a set element of the result set. In other words, the result set contains graphs, but no sets of graphs.

graph $(< arg_i >)$ =

<u>if</u> $< arg_i > = (< f_{i1} >, \ldots, < f_{ik} >)$

<u>then</u>

$\{$ graph$(< f_{i1} >)$ $\cdots$ graph $(< f_{ik} >)$ $\}$

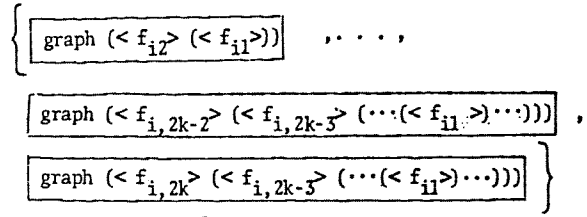
<u>else if</u> $< arg_i > = f_i\ (< arg_{i1} >, \ldots, < arg_{ik} >)$

<u>then</u>

$\{$ graph $(< arg_{i1} >)$ $\cdots$ graph$(< arg_{ik} >)$ $\}$

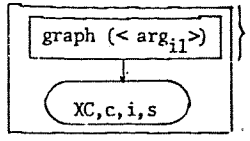
<u>else if</u> $< arg_i > = [< f_{i1} > : < f_{i2} >, \ldots, < f_{i,2k-3} > : < f_{i,2k-2} >,$
<u>true</u> : $< f_{i,2k} >]$

<u>then</u>



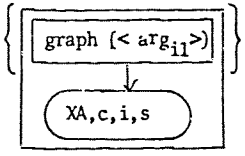
else if $< arg_i > = XC^s_c\ (< arg_{i1}>)$

then



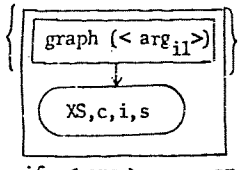
else if $< arg_i > = XA^s_c\ (< arg_{i1}>)$

then



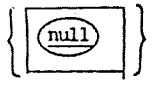
else if $< arg_i > = XS^s_c\ (< arg_{i1}>)$

then



else if $< arg_i > = \sigma_j$ or $< arg_i > = f_i$

then



As an example of the use of "graph," Figure 5 shows the value yielded by "graph" when applied to the specification fragment in Figure 4.

The "connect" function maps a preliminary graph into one without null nodes, in which sets of nodes, all of which represent the same exchange evaluation, are merged into single nodes. This is done by checking superscripts.

"Connect" simply eliminates all null nodes, and merges all sets of nodes with same type, class, and superscript. When a null node is eliminated, all its input arcs are joined to all its output arcs, to preserve precedence constraints. When a set of nodes is merged, a single node representing the whole set is formed. It has the common type and class, and a pointerset containing the pointers of all merged nodes. It has the predecessors of all merged nodes, and the successors of all merged nodes.

Since "connect" operates only on one preliminary graph, the characteristic set of graphs for a process $(< \Sigma_i >, < f_i >)$ is $\{g : \exists\, h \in graph\ (< f_i >),\ g = connect\ (h)\}$. Figure 6 shows the set of graphs for the process whose

$< f_i > =$

(

$f_{i1}(f_{i11}\ (XC_1^1\ (f_{i1111}), \sigma_1)\ ,\ XC_2^1\ (f_{i121}(XC_1^1(f_{i12111}),\ \sigma_1)))$

,

$[f_{i21}\ (XC_1^1\ (f_{i2111})) : XC_3^1\ (f_{i221})$

,

$f_{i23}(XC_1^1(f_{i2311})) : XC_4^1\ (f_{i241})$

,

true : σ_2

]

)

Figure 4. A specification fragment.

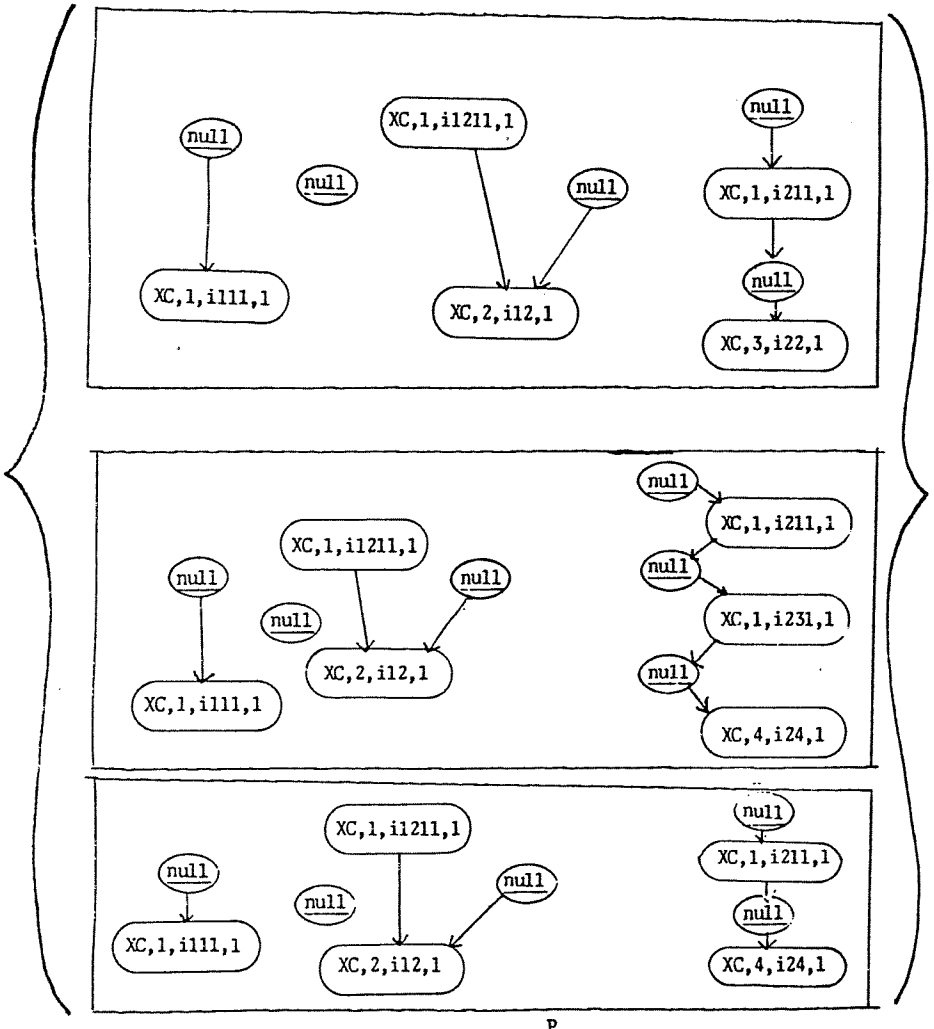


Figure 5. A member of 2^{G^P}.

successor function is specified in Figure 4. Definition of an actual algorithm to find this set should be straightforward.

One of the most important properties of interaction is determinacy: an exchange class is said to be determinate if for every instance of an exchange

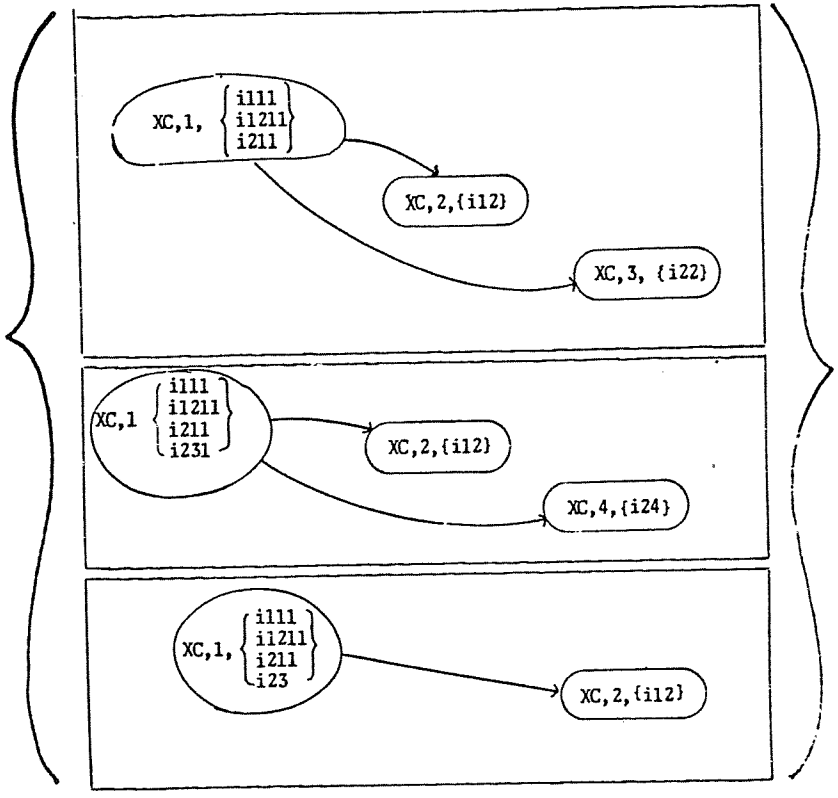


Figure 6. A member of 2^G.

of that class in the specification, there is at most one other instance with which it can match. The second analysis algorithm uses the graph sets produced by the first to define simple sufficient conditions for establishing determinacy.

Basically, the idea is to rule out the obvious non-matches for an exhange: itself, those of the wrong type, those of the wrong class, and predecessors or successors in the same process. If the remaining set of exchanges in a specification is called a "potential set" for the exchange, then a detectably deterministic class is one in which no member has a potential set with a size greater than one.

Why is an exchange ruled out as a match for itself? Each exchange in a specification is instantiated by a sequence of evaluations (during different steps of the same process), but there is a strong precedence constraint separating members of the sequence. The same is true, of course, of exchanges in the same process step with a predecessor/successor relationship. We note for future reference that any transformation which purports to preserve determinacy, as detected by this analysis, must also preserve these precedence constraints.

Let N be the set of all nodes appearing in all graphs produced by the preceding algorithm applied to a specification, and let $n_{ijk} = (t_{ijk}, c_{ijk}, p_{ijk})$ be the $k^{\underline{th}}$ node in the $j^{\underline{th}}$ graph of the $i^{\underline{th}}$ process. Exchanges in the specification are uniquely identified by pointer-sets, and the predicate deterministic $(\hat{c}) \equiv [\, \forall\, p_{ijk} \ni c_{ijk} = \hat{c}\ ,\ |\ \text{potentials}\ (p_{ijk})\ | \leq 1]$ decides determinacy of a class $\hat{c}$. The potential set is found using: potentials $(p) = \{\hat{p} : [\, p \neq \hat{p}\,] \,\&\, [\, \exists\, n_{ijk} = (t_{ijk}, c_{ijk}, p_{ijk}), n_{i'j'k'} = (t_{i'j'k'}, c_{i'j'k'}, p_{i'j'k'}) \in N \ni p_{ijk} = p,\ p_{i'j'k'} = \hat{p} : [c_{ijk} =$

$c_{i'j'k'}\,] \,\&\, \neg\, [\, t_{ijk} = t_{i'j'k'} = XA\,] \,\&\, \neg\, [\, t_{ijk} = t_{i'j'k'} = XS\,] \,\&\, [[\, i = i'\,] \vee [[\, j = j'] \& \neg [n_{ijk} \subseteq \text{pred}\ (n_{i'j'k'})]] \,\&\, \neg\, [n_{i'j'k'} \subseteq \text{pred}\ (n_{ijk})]]]]]\},$[4] where pred(node) is the set of all predecessors of a node in a directed graph. As before, a straightforward graph algorithm is implied by these expressions.

All deterministic classes can be split into new classes, each of which has only two exchanges (XC/XC or XS/XC), without changing the semantics of the specification. This "canonical form" encodes the determinacy explicitly, making detection trivial, since all potential matches but one can now be ruled out on the basis of the first two clauses in "potentials." Henceforth we will assume that all specifications use this canonical form.

C. A Particular Definition of Equivalence

This section presents the formulation of equivalence which seems most suitable for the problem at hand, i.e. optimizing transformations on closed systems.

In a closed system, the environment which uses the open digital system being developed must be represented approximately as another open digital system, as shown in Figure 7. There is no apparent loss of generality in requiring that the dividing line be drawn between (rather than through) processes, since digital processes cannot be synchronous with the continuous processes being approximated by the "environment" digital processes.

An optimizing transformation operates on the "subsystem" without changing the "environment." To be correct, it must preserve the functional properties

[4] There may be more than one node in N with the pointerset p, but since all represent the same exchange, all have the same type, class, and process membership.

of the subsystem as seen by the environment. Our formal definition of equivalence must be designed so that transformations from systems to equivalent systems are correct by this standard.

An equivalence relation on a set of asynchronously interacting processes can actually be defined without explicit mention of the interaction mechanism. Let S be the system whose specification is $((\Sigma_1,f_1), (\Sigma_2,f_2),\ldots(\Sigma_n,f_n))$, where Σ_i is a state space and f_i is a (possibly non-deterministic) successor function. Then formally S is the set of all (possibly infinite) sequences $s_0,s_1,s_2,\ldots$ such that $s_i \in \prod_{j=1}^{n} \Sigma_i$, $\forall_i$, and, $\forall_i$, if $s_i = (\sigma_1,\sigma_2,\ldots,\sigma_n)$, $s_{i+1} = (\sigma_1',\sigma_2',\ldots,\sigma_n')$, there exists a k such that $\sigma_k' = f_k(\sigma_k)$, $\sigma_j' = \sigma_j$ if $j \neq k$.

Let p_J be the function which projects a tuple onto some of its elements, namely those whose indices are contained in the integer set J. Thus if t is an n-tuple, and J contains k integers between 1 and n inclusive, the p_J will project the n-tuple onto a k-tuple.

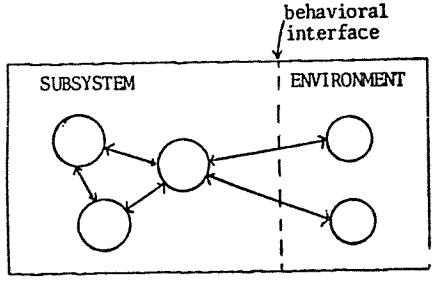


Figure 7. Parts of a closed system

Let $P_J(S)$ be $\{t : \exists\, q \in S,\ q = s_0,s_1,s_2,\ldots,t = t_0,t_1,t_2,\ldots,$ and $\forall_i,$ $t_i = p_J(s_i)\}$. Then we can say that S is equivalent to S' with respect to J environment processes, or $S \underset{J}{\equiv} S'$, if $P_J(S) = P_J(S')$.

This definition says that two systems are equivalent if their environment subsystems, viewed as systems, contain the same computations. Since the "meaning" of a closed system is presumably what the environment part can do, this simple definition seems adequate and appropriate.

D. Preliminary Observations

Now that we know what equivalence is, we must ask ourselves what kinds of transformations can be proven to preserve equivalence, and are useful as optimizations.

A computation of a system can be viewed as a large number of primitive function evaluations, governed by precedence constraints. Some of the constraints are the "natural" ones determined by the functional structure, e.g. a function cannot be evaluated until its arguments have been. Others are the "artificial" constraints induced by process membership, i.e. that all function evaluations belonging to one step of one process must be completed before any evaluations belonging to the next step of that process begin.

We could not optimize by changing the primitive functions (or, by implication, their data structures), or their "natural" precedence constraints, without introducing some form of axiomatic definitions of primitives with which to carry out the equivalence proofs. There seems to be little advantage in doing this, because:

(1) the proofs would still be impossible in general and difficult in particular;

(2) such changes can be properly termed evolutions, and done without proof of equivalence.

Thus, the possible changes remaining concern the number of instances of a function in the specification, the "artificial" precedence constraints, etc. These are important examples because the first concerns quantity and utilization of implementation resources, and the second concerns the distribution of computations over an asynchronous network. Section V.A contains the complete classification of transformations discussed in this report.

In the proofs found in Part V, the environment consists of all processes in the system whose specifications do not change. This is not necessary for equivalence with respect to the smaller "real" (i.e. user-defined) environment, but is a sufficient condition which holds, we believe, for all practical proofs.

IV. A FUNDAMENTAL THEOREM

In this section a theorem will be proved, establishing some sufficient conditions for the preservation of equivalence under a transformation. These conditions are tailored to the anticipated optimizations; subsequent proofs about specific transformations will refer to this theorem rather than to the definition of equivalence.

We remind the reader that transformations are assumed to be from valid specifications to valid specifications. This means that it is not necessary for us to consider here the possible introduction of inconsistencies, such as exchange blockages, by the transformations - because these do not appear in valid specifications.

Since transformations are applied to specifications rather than systems, it is first necessary to extend the definition of equivalence to specifications. A specification T generates a system S under a mapping I called an interpreter (which will be specified in the near future). Equivalence of specifications is simply defined as $T \underset{J}{\equiv} T'$ if and only if $I(T) \underset{J}{\equiv} I(T')$.

To learn more about equivalence, we must decompose I. Let $i : \mathcal{T} \times \mathcal{S} \times \mathcal{R} \to \mathcal{S} \times \mathcal{S}$ be an interpretation function such that $I(T) = \{q : \exists s \in \mathcal{S}_T, \exists r \in \mathcal{R}, i(T,s,r) = (g,q)\}$, where $\mathcal{S}_T$ is the state space $\prod_{k=1}^{n} \Sigma_k$ of T. $\mathcal{T}$ is the set of all specifications, $\mathcal{S}$ is the set of all states of specifications, and $\mathcal{R}$ is a set whose members encode choices about relative rates, exchanging matching etc. in such a way that all time-dependent events are determined. $\mathcal{G}$ is the set of all computations (state sequences), and $\mathcal{H}$ is a set of directed graphs.

The nodes of a graph in $\mathcal{H}$ represent primitive function evaluations, under non-unique definitional names, and the arcs represent functional precedence constraints, i.e. those induced by the necessity to evaluate all the arguments

of a function before the function itself (the evaluation constraints on primitives in a selection construction are also included here). In addition, two exchange evaluations which match are joined by a double-headed arrow.

The "meaning" of i is as follows: For a particular specification, a state and an encoding of time-dependent decisions uniquely determine what will happen when the specified system is started with that initial state. "What will happen" can be, and is, expressed in two different ways: as the resultant computation, and as an "execution trace" graph of what functions were evaluated and what necessary logical relationships these evaluations had to one other.

Fundamental Theorem:

Let $T = ((\Sigma_1,f_1),\ldots,(\Sigma_n,f_n))$ and $T' = ((\Sigma_1', f_1'),\ldots,(\Sigma_{n'}',f_{n'}'))$ be specifications such that $p_J(T) = p_J(T')$ for some index set J. Let $K = \{1,2,\ldots n\} - J$, $K' = \{1,2,\ldots n'\} - J$, and let there be a permutation $m : \prod_{k\in K}(\prod_{\ell}\Sigma_{k\ell}) \to \prod_{k\in K'}(\prod_{\ell}\Sigma'_{k\ell})$.

Then $T \underset{J}{\equiv} T'$ if for all pairs (s,s') such that $s \in \prod_{k=1}^{n}(\prod_{\ell}\Sigma_{k\ell})$, $s' \in \prod_{k=1}^{n'}(\prod_{\ell}\Sigma_{k\ell})$, $p_J(s) = p_J(s')$, and $p_K(s) = m^{-1}(p_{K'}(s'))$:

(1) $G = \{g : \exists r,\ i(T,s,r) = (g,q)\} = G' = \{g : \exists r,\ i(T',s',r) = (g,q)\}$;

(2) if f is a node (function evaluation) in $\hat{g} \in G, G'$ such that one of its arguments is an initial state component, and c, c' are the projections which select the used component out of $\prod_{k=1}^{n}(\prod_{\ell}\Sigma_{k\ell})$, $\prod_{k=1}^{n'}(\prod\Sigma_{k\ell})$, respectively, then $c(\prod_{k=1}^{n}(\prod_{\ell}\Sigma_{k\ell})) = c'(m(\prod_{k=1}^{n}(\prod_{\ell}\Sigma_{k\ell})))$.

Proof:

We must show that for all $\hat{g} \in G,G'$, $i(T,s,r) = (\hat{g},q)$ and $i(T',s',r') = (\hat{g},q')$ for some $r,r' \Rightarrow p_J(q) = p_J(q')$.

We know by assumption that the processes indexed by J are identical in the two systems, and also receive identical initial values. Thus the only way that their computations could differ would be if they received different values, through exchanges, from the processes indexed K and K', respectively.

There is no way, however, that any function evaluation can have different arguments or yield a different result in the two computations, because:

(1) the common graph $\hat{g}$ shows that the same primitives were evaluated and that the argument - passing structure was the same (including that involving exchanges, because of the identical matches);

(2) the theorem specifies that the initial values received by primitives in both cases are the same. □

V. EQUIVALENCE-PRESERVING, OPTIMIZING TRANSFORMATIONS

A. Classification

In this part a set of optimizing transformations will be given, with proofs that they preserve equivalence. Although the equivalence relation itself is symmetric, the transformations and accompanying examples will be motivated by the design theory as a whole, and therefore go in the direction which seems most useful. Should transformations in the other direction be required, it should be easy to provide them.

As noted in III.D, a computation consists of a number of primitive function evaluations (implying the data structure) linked by precedence constraints, and we do not intend to include operations on primitive functions and "natural" precedence constraints in our optimizations. All this leaves us to manipulate are:

(1) "artificial" precedence constraints originating in process membership rather than argument-result structures;

(2) the existence of duplicate or superfluous primitives in the specification.

Subdividing these gives us our four possible kinds of optimizing transformations:

(1a) changes to the scope of precedence constraints associated with the endpoint of a process step (see Figure 8);

(1b) changes to the frequency of precedence constraints associated with the endpoint of a process step (see Figure 8);

(2a) addition or deletion of duplicate primitives;

(2b) addition or deletion of logically superfluous primitives.

B. Transformations of Type 1a: "Process Splitting"

Although type 1a refers to splitting one process into several or joining several processes into one, our design theory strongly favors application of the former. This is because design is most fundamentally a process of elaboration, and an unelaborated specification exhibits less structure that can be exploited with parallelism than an elaborated one. In other words, designs begin with one or a few processes; as these are elaborated, new structures arise which can be split off as separate processes. Thus a "process splitting" transformation is the fundamental tool by which computations are distributed. Such a transformation is given here.

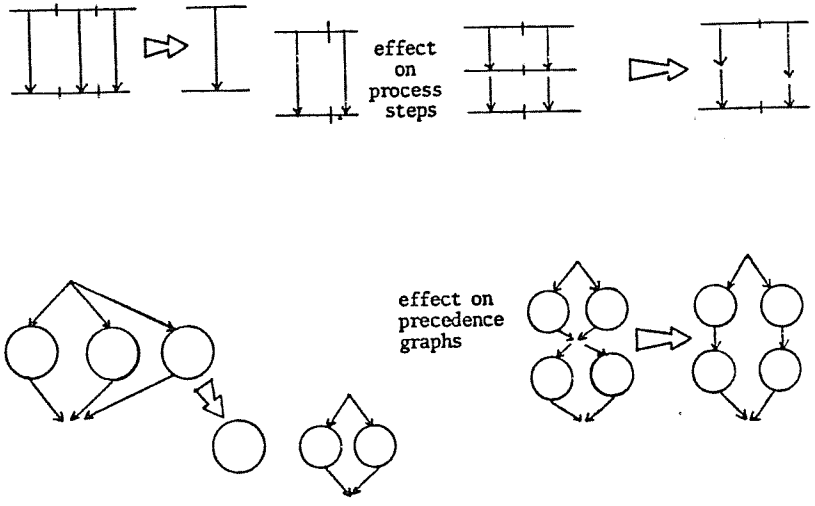


Figure 8. Transformations of type 1.

Transformation 1a splits one process of a system into two. Any of the state components of the original process, and of the functional structure, can be removed to form the second process. Arguments and values are communicated between the two processes using deterministic XC/XC interactions.

Transformation 1a:

Let $((\Sigma_1,f_1),\ldots(\Sigma_{k1} \times \Sigma_{k2} \times\ldots\times \Sigma_{km}, (<f_{k1}>,<f_{k2}>,\ldots,<f_{km}>)),$ $\ldots, (\Sigma_n,<f_n>))$ be a system specification, and let P,Q be index subsets of N and N+ , respectively. This means we want to form a new process containing all Σ_{kp} , $p \in P$, and all $<f_{kq}>$, $q \in Q$.

The new state space of process k will be $\prod_{j \notin P} \Sigma_{kj}$, and the state space of the new process n+1 will be $\prod_{j \in P} \Sigma_{kj}$. The functions appearing in the specification of k will be $\{<f_{kj}> : j \notin Q\}$, and those appearing in n+1 will be $\{<f_{kj}> : j \in Q\}$. Pieces of the functional structure can stay intact as much as this index-set partition allows. The only restrictions on Q are that no substructure of a selection structure can be separated, and exchanges which are "equivalenced" by superscripting to the same evaluation cannot be separated.

For every component of a state space, there must be a function which generates its next value. If Σ_{kj} and $<f_{kj}>$ are left in the same process (k or n+1), this is no problem. But if they are separated, then Σ_{kj} must have a new component successor function $XC_i(\underline{null})$ to get its value, where i is a unique class identifier. How the value will be transmitted is shown below.

In fact, any time a function is missing an argument because that argument is evaluated by a function in the other process, $XC_i(\underline{null})$ is substituted (each time a new unique i is used).

The final problem is to invoke functions whose values are needed in

the other process, and to pass on the generated values. Suppose process k owes m uses of such functions to other processes. Then the successor function of k is

$$\text{proj}_{\ell}^{\ell+m}(\underbrace{<f_{kj_1}>,<f_{kj_2}>,\ldots<f_{kj_\ell}>}_{j_p \in P},\ XC_{i_1}(<f>),\ XC_{i_2}(<f'>),$$

$$\ldots\ XC_{i_m}(<f^{\overbrace{111\ldots1}^{m}}>))\ ^{5},$$

where the primed f's are the functions to be invoked, and the i's are the exchange classes defined at the point of use of these functions.

Example: Distributing a Computation

A process has been developed with a state space $\Sigma_1 \times \Sigma_2 \times \Sigma_3$ and a successor function (f_1,f_2,f_3), where $f_1 = e(\sigma_1,\sigma_2)$, $f_2 = g(\sigma_2,\sigma_3)$, and $f_3 = h(\sigma_3)$. It has been decided that the computation done by this process should be distributed over two nodes of a network, one of which contains Σ_1, Σ_2, and f_1 , while the other contains Σ_3, f_2, and f_3 .

Then the resultant processes, after applying Tranformations 1a with $P = \{2,3\}$, $Q = \{3\}$, are:

$(\Sigma_1 \times \Sigma_2,\ \text{proj}_2^3(e(\sigma_1,\sigma_2),\ XC_x(\underline{null}),\ XC_y(\sigma_2)))$;

$(\Sigma_3,\ \text{proj}_1^2(h(\sigma_3),\ XC_x(g(XC_y(\underline{null}),\sigma_3))))$.

Example: Isolating a Function

A process has been developed with state space Σ_{k1} and a successor function $(f_{k1}) = e(g(h(\sigma_{k1})))$. The function h will run much better on a special processor, and so it is decided that this function should be evaluated in a separate process which will then be implemented with specialized hardware.

The resultant processes, after applying Transformation 1a with $P = \{\ \}$,

---

5 The function $\text{proj}_{\ell}^{\ell+m}$ projects an $(\ell+m)$- tuple onto its first ℓ components.

Q = {k111}:

$(\Sigma_{k1}$, $proj_1^2$ (e(g(XC_x(null))), $XC_y(\sigma_{k1})$));

$(\Sigma_{null}$, $proj_1^2$ (null, XC_x(h(XC_y(null))))).

Repeated applications of this transformation can produce as many new processes as desired. Note that the possibility of exchange blockage does not arise because the invocations of "missing" functions appear in a vector unconstrained by precedence.

Theorem 1a:

Let T be a specification with n processes, and let T' be the result of applying Transformation 1a to process k of T with index sets P and Q. Then $T \underset{J}{\equiv} T'$, J = {1,2,...,k-1,k+1,...,n} unless:

(1) the two new processes are completely independent, and each has an XS which interacts with an XC in the environment, or

(2) Q is such that it includes an XA or XS , while excluding an XA or XS , respectively, of the same class.[6]

Proof:

The mapping between states s of T and states s' of T' is the obvious permutation, and the transformation ensures that components are used as arguments to the same functions as before. Thus the Fundamental Theorem tells us that all we need to show is that for initial states s , s' = m(s) , there are relative rates for which T and T' exhibit the same function evaluation, argument passing, and exchange matching structures.

The transformation is designed to ensure that the function evaluations and argument passings are preserved. Since it is not possible to split up a selection construction, all "new" exchanges are unconditionally evaluated,

[6] Of course, the two examples meet the conditions of this theorem.

and the two altered processes either run in a "loose lockstep" (one can never get more than a fraction of a step ahead of the other) or are independent. Thus all that needs to be checked is the exchange matching.

For deterministic interactions, the only possible variation in matches is which XS in a sequence an XC matches. The loose lockstep of the split processes will provide the same range of possibilities to the environment as would the original process.

For non-deterministic interactions, the possible configurations for exchange of a certain class within the original process are:

(1) one XA, XS, or XC;

(2) multiple XA's;

(3) multiple XS's.[7]

One of anything will look the same to the environment, regardless of which new process it goes to. If multiple XS's or XA's are split by the transformation the environment might see a difference, but this case is precluded by the theorem. □

C. Transformations of Type 1b: "Scheduling"

Although type 1b refers to increasing parallelism within a process by making the steps longer, or decreasing parallelism by making the steps shorter, the latter is much more likely to be called for than the former. Automatic detection of parallelism is likely to yield no more than trivial results, especially in a context in which specifications are being developed top-down - for the best time to recognize that two computations can be done

[7] The reasons are as follows. No non-deterministic class need have more than one XC: they are either multiple XA/XC or possibly-multiple XA/possibly-multiple XS. And it is easily shown that putting members of a non-deterministic class which could match (as multiple XA's and XS's cannot) allows the possibility of blockage.

in parallel is when elaboration first differentiates them as separate computations!

Scheduling, on the other hand, will always be needed as we move from sufficient-resource models to scarce-resource models. Equivalence-preserving transformations which introduce precedence constraints are the way to introduce this to our formal design method.

Transformation 1b turns a single step of a process into two, with the values of the "phase one" evaluations being stored in the state between steps, and used as the arguments for the evaluations in "phase two".

Transformation 1b:

Let $(\Sigma_{k1} \times \Sigma_{k2} \times \cdots \times \Sigma_{kn}, (< f_{k1} >, < f_{k2} >, \ldots < f_{kn} >))$ be a process specification and let Q be an index set, $Q \in k \oplus N+$, such that if $k\ell \in Q$, then $k \ell m \notin Q$, $\forall m$. The elements of Q represent the "cutoff points" between the first and second steps, and are restricted so that two exchanges which are "equivalenced" to the same evaluation by superscripts cannot be separated. Also, a "cutoff" cannot come between substructures of a selection structure.

The state space of the new process is $\{1,2\} \times \Sigma'_{k1} \times \Sigma'_{k2} \times \cdots \times \Sigma'_{kn}$, where Σ'_{kj} is the union of Σ_{kj}, and the cross product of Σ_{kj} and the ranges of all $< f_{kj\ell} >$ such that $kj\ell \in Q$. $f_{k0} : \{1,2\} \to \{1,2\}$ is:

$f(\sigma_{k0}) = [\sigma_{k0} = 1 : 2 , \sigma_{k0} = 2 : 1]$,

so that the first state component is the phase counter.

In the new process, $< f_{kj} >$ takes the following form:

$[\sigma_{k0} = 1 : (\sigma_{kj}, < f_{kj\ell} >)_{kj\ell \in Q}$

,

$\sigma_{k0} = 2 : < f'_{kj} >$

],

where $< f'_{kj} >$ differs from $< f_{jk} >$ in that all $< f_{kj\ell} >$, $kj\ell \in Q$, are replaced by $\text{proj}^{y+1}_{x+1}(\sigma_{kj})$, where y is the number of all $kj\ell \in Q$, and x is the index of this temporary value in the y-vector of them.

If all or none of $< f_{jk} >$ is to go into phase 1, then an identity transformation must be inserted in phase 2 or 1, respectively.

Example: Scheduling

A specification contains this process:

$(\Sigma_{k1} \times \Sigma_{k2} \times \Sigma_{k3}, ([\sigma_{k1} = g(\sigma_{k2}) : h_1(\sigma_{k1}), \underline{\text{true}} : h_2(\sigma_{k1})], (s_1(\sigma_{k2}), s_2(\sigma_{k2}), s_3(\sigma_{k2})), t_1(t_2(t_3(\sigma_{k3}))))$. Since the resources are not available to do all these function evaluations in parallel, nor is it necessary to meet real-time deadlines[8], it is decided to apply Transformation 1b with $Q = \{k1, k22, k23, k311\}$. The result is this new process:

$(\{1,2\} \times \Sigma_{k1} \times \Sigma_{k2} \cup (\Sigma_{k2} \times \mathcal{R}(s_2) \times \mathcal{R}(s_3)) \times \Sigma_{k3} \cup (\Sigma_{k3} \times \mathcal{R}(t_3))$

,

$[\sigma_{k0} = 1 : 2 , \sigma_{k0} = 2 : 1]$

,

$[\sigma_{k0} = 1 : [\sigma_{k1} = g(\sigma_{k2}) : h_1(\sigma_{k1}), \underline{\text{true}} : h_2(\sigma_{k1})] , \sigma_{k0} = 2 : \sigma_{k1}]$

,

$[\sigma_{k0} = 1 : (\sigma_{k2}, s_2(\sigma_{k2}), s_3(\sigma_{k2})),$
$\sigma_{k0} = 2 : (s_1(\text{proj}^3_1(\sigma_{k2})), \text{proj}^3_2(\sigma_{k2}), \text{proj}^3_3(\sigma_{k2}))]$

,

$[\sigma_{k0} = 1 : (\sigma_{k3}, t_3(\sigma_{k3})),$
$\sigma_{k0} = 2 : t_1(t_2(\text{proj}^2_2(\sigma_{k3})))]$

),

[8] In fact, we think it likely that the way to meet speed and cost requirements is to start with the most parallel design, then reduce its performance by scheduling, as time permits, to meet cost limits.

where $\mathcal{R}$ now signifies the range of a function. The effect of the transformation is shown pictorially in Figure 9.

Theorem 1b:

Let T be a specification with n processes, and let T' be the result of applying Transformation 1b to process k of T with index set $Q \in k \circ N+$. Then $T \underset{J}{\equiv} T'$, $J = \{1,2,\ldots,k-1,k+1,\ldots n\}$, unless:

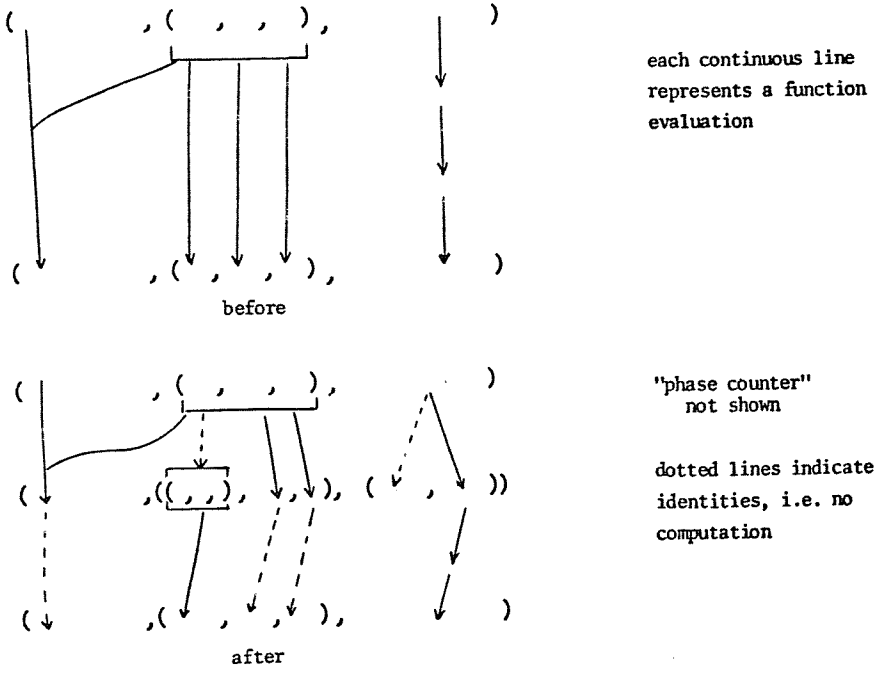


Figure 9. Transformation 1b in action.

(1) there are at least two XS's in the specification of k which interact deterministically with the environment, and one is put in phase one while another goes to phase two, or

(2) there are at least two XA's or XS's, respectively, of the same class and interacting non-deterministically with the environment, and one is put in phase one while another goes to phase two.

Proof:

The mapping from states s of T to states s' of T' is $m(\sigma_{k1}, \sigma_{k2},\ldots,\sigma_{kn}) = (1,\sigma_{k1},\sigma_{k2},\ldots,\sigma_{kn})$. We need not concern ourselves with the fact that phase two states have no inverse mapping, because equivalence is defined on a projection in which states of process k do not appear. The transformation ensures that state components are used as arguments to to the same functions as before, and that the same functions are evaluated. Nothing remains to be altered except exchange matching.

The only effect on the variety of exchange matches would be to separate exchanges into separate phases, thus constraining one to follow the other, always. But we have precluded all cases in which such a difference would be visible to the environment. □

D. Transformations of Type 2a: "Resource Sharing"

Type 2a refers to the addition or deletion of duplicated primitives. Addition would mean adding extra resources to get a job done faster, a straightforward transformation. It is unlikely to be used in our method, however, because development from sufficient resource specifications to scarce resource specifications seems indicated. In such a development process, the standard transformation would be to delete duplicated primitives, i.e. share resources.

Before we can claim to be "sharing resources", of course, we must establish what a resource is. Investigation of this topic is underway.[9] In the meantime, we will restrict ourselves to a special case which is clearly understood: the type of process, containing a single function remotely invoked, created by Transformation 1a in the "Isolating a Function" example.

The transformation operates on a specification in which there are several such processes, each housing the same function. It reduces these to one multiplexed process.

Transformation 2a:

Let T be a specification, and let P be an index set of processes in it such that if $p \in P$, then process p has the form:

$(\Sigma_{\underline{null}}, \text{proj}_1^2(\underline{null}, XC_x(g(XC_y(\underline{null})))))$.

The function g must be the same for all such processes, but the exchange classes x and y will be different in each case.

Transformation 2a creates a system in which all but one of these processes is gone. Let the one remaining have exchange classes x_0 and y_0. Then in the remaining processes of the specification, any occurrence of XC_{x_1}, x_1 being a class which is used as an x (above) in some process $p \in P$, is replaced by XC_{x_0}. Similarly, any XC_{y_1} is replaced by XA_{y_0}.

Example: Resource Sharing

Two applications of Transformation 1a have produced this specification:

$((\Sigma_1, \text{proj}_1^2(f(XC_x(\underline{null})), XC_y(g(\sigma_1))))$

,

$(\Sigma_{\underline{null}}, \text{proj}_1^2(\underline{null}, XC_x(h(XC_y(\underline{null})))))$

,

$(\Sigma_3, \text{proj}_1^2(s(XC_u(\underline{null})), XC_v(t(\sigma_3))))$

,

$(\Sigma_{\underline{null}}, \text{proj}_1^2(\underline{null}, XC_u(h(XC_v(\underline{null})))))$

).

[9] By D. R. Fitzwater, as part of the study of operating systems.

The next step of development is to share the resource which is the implementation of h, i.e. apply Transformation 2a with $P = \{2,4\}$. It yields:

$((\Sigma_1, \text{proj}_1^2(f(XC_x(\underline{null})), XA_y(g(\sigma_1))))$

,

$(\Sigma_{\underline{null}}, \text{proj}_1^2(\underline{null}, XC_x(h(XC_y(\underline{null})))))$

,

$(\Sigma_3, \text{proj}_1^2(s(XC_x(\underline{null})), XA_y(t(\sigma_3))))$

).

Theorem 2a:

Let T be a specification with n processes, and let T' be the result of applying Transformation 2a to T with index set P. Then $T \underset{N\text{-}P}{\equiv} T'$.

Proof:

By inspection. □

E. Transformations of Type 2b : "Identities"

Type 2b refers to the addition or deletion of logically superfluous primitives, i.e. identity functions. Since it is not yet clear at what points in the development process a need for this might occur, we will disucss a few examples only.

The processes:

(1) $(\Sigma,\ f(XC_i(\sigma)))$

(2) $(\Sigma, [XS_i^1(\sigma) \neq \sigma : f(XS_i^1(\sigma)),\ \underline{true} : \sigma])$

are logically equivalent in an environment with a single XC_i, regardless of whatever side effects evaluation of f may have, because f is evaluated in both cases only when the environment XC_i has an argument to exchange. The difference is that between these times, (1) waits while (2) cycles on an identity transformation, i.e. busy waits. Form (1) might seem superior in most cases, but (2) corresponds better to a physical peripheral device, for example.

Another possible transformation would be to introduce "delay element" on a communication path. By this we mean a process which acts as an intermediary between two exchanges which would have matched each other, matching both, introducing some delay, then matching a second exchange evaluation from each side to pass the information. The purpose of this would be to allow explicit modeling of communication delays. It is also discussed in [ZF].

In general, insertion and deletion of identity transformations should be easily done and proved equivalence-preserving, as needed.

VI. COMPARISONS TO OTHER EQUIVALENCE STUDIES

Due to the difficulty of the equivalence problem when approached in the wrong context (see Part VII), there is little work which is directly comparable to this. In [Kn] Knuth mentions the idea of an automated lab for making optimizing transformations on programs - in much the same spirit as this work - without offering any particular hope of its realization.

Axiomatic proof techniques, originally developed for sequential programs ([Ho]), are now being extended to so-called "parallel programs", which are models limited to multiprogrammed implementations ([Ke], [OG], [La]). These can be considered related in the sense that a proof of correctness is a statement of equivalence to some standard of correctness. But there is actually a very significant difference, because they are dealing with arbitrary cases in a way that requires a great deal of human ingenuity. We are dealing with highly structured cases algorithmically. Comparisons of their relative utility for very large systems must inevitably favor the latter.

The closest work to that presented here is reported in [Ri] and [Za]. In both cases equivalence of open systems is formally defined. But because of the inherent complexity of these general, relative-rate-dependent definitions, efforts to use them productively are not particularly successful. The contrast between our present work and these will be discussed futher in Part VII.

VII. CONCLUSIONS

In this report we have characterized exhaustively one branch of our proposed design method, and shown significant, useful results within that branch. As far as we know, non-trivial results on system equivalence are a unique achievement.

Just as important as the success of this work are the reasons for its success, because they reflect on the proposed design theory as a whole. We believe that this work has succeeded where [Za], for instance[10], failed, because:

(1) [Za] used open systems, and we are now using closed systems. It is much easier to prove the equivalence of two subsystems with respect to a particular environment, than with respect to any environment. The different forms of equivalence relations in [Za] were an attempt to restrict the operative environment, but just weren't good enough.

(2) [Za] was done without assumption of any knowledge about what source systems would be like or what target systems would be needed. Now our design theory gives us strong guidance about what transformations will be needed, greatly restricting the problem domain.

(3) The fatal flaw of the notion of equivalence in [Za] was that computations and the definition of equivalence itself were so completely dependent on relative rates, resulting in a combinatorial explosion of complexity. This problem has plagued all other research on the subject, of course.

[10] This is the best possible comparison because the goals are known to have been identical, and differences of results cannot be attributed to differences in the quality of personnel!

Exchange functions, with their amazing robustness,[11] promise to be the best tool for coping with relative rates to come along yet.

Two aspects of this work have a disappointing appearance. The first is that there are so few transformations given here, and that they are so simple conceptually. In fact, this simplicity is another sign of success.

The reason that so many possible optimizations come to mind for programs, for instance, is that programs, with their procedural structure, fixed set of primitives, etc., force a high degree of over-sepcification on the programmer. He must make many decisions about data and control structures before it is possible to understand the implications of these decisions for resources and performance. It is not at all surprising that he would like to revoke some of these decisions when their implications are understood. But by then it is usually too late, because the impact of changes on the rest of a highly developed system cannot be assessed - hence the dismal record of attempts at anything but code (very local) optimization.

In our design method, by contrast, it is never necessary to over-specify. What was not specified prematurely[12] need not be optimized later. This explains the small number of transformations in our set. But the transformations we have are just the ones we need to carry out our development steps - at least the ones we know about so far.

The other troublesome aspect is the formalism, the notation for expressing transformations and carrying out proofs. It is crude, and far behind our intuitive understanding of the subject.

The problem is simply that the notation here is premature. At the time

[11] For instance, a deterministic XC/XC interaction will do the same thing regardless of any variation in relative rates.

[12] This is not meant to rule out iterative design, which will always be necessary in some cases. Iterative design leads to evolution steps (see II.C).

of this writing we have just finished designing the syntax of the specification language. Formalism on which to base algorithms needs to be developed now in approximately this order:

(1) the form of the design data base;

(2) completeness and consistency checks;

(3) analysis of exchange blockage;

(4) equivalence-preserving transformations.

There seems to be no reason why these formalizations cannot be carried out within an appropriate time frame.

In conclusion, the parts of our proposed design method which rely on algorithmic equivalence-preserving transformations seems to be entirely feasible. At the level of our present understanding, no complexity issues are raised beyond those of handling a large design database and performing consistency checks on it (including exchange analysis). Furthermore, the fact that we can make progress so easily on a problem that has been considered too hard even to attempt in other contexts, is a strong recommendation for our design theory as a whole.

Acknowledgement

This work would not have been possible without the leadership and collaboration of D. R. Fitzwater.

REFERENCES

[Fi] Fitzwater, D. R. "The Formal Design and Analysis of Distributed Data-Processing Systems," University of Wisconsin Computer Sciences Department TR-295, 1977.

[FZ] Fitzwater, D. R., and Zave, Pamela. "The Use of Formal Asynchronous Process Specifications in a System Development Process," Sixth Texas Conference on Computing Systems, 1977.

[Ho] Hoare, C.A.R. "An Axiomatic Basis for Computer Programming," CACM 12, October 1969.

[Ke] Keller, Robert M. "Formal Verification of Parallel Programs," CACM 19, July 1976.

[Kn] Knuth, Donald E. "Structured Programming with go to Statements," Computing Surveys 6, December 1974.

[La] Lamport, Leslie. "Proving the Correctness of Multiprocess Programs," IEEE Transactions on Software Engineering SE-3, March 1977.

[OG] Owicki, Susan, and Gries, David. "Verifying Properties of Parallel Programs: An Axiomatic Approach," CACM 19, May 1976.

[Ri] Riddle, William E. "An Approach to Software System Modelling, Behavior Specification, and Analysis," To appear in CACM.

[Za] Zave, Pamela. "Functional Equivalence of Parallel Processes," University of Maryland Computer Science Department TR-439, 1976.

[ZF] Zave, Pamela, and Fitzwater, D. R. "Specification of Asynchronous Interactions Using Primitive Functions," University of Maryland Computer Science Department TR-598, 1977.

Appendix E - Index to Definitions in Section 3

abstraction 3.3.4

abstractions of discrete processes 3.1.1

algorithmic procedure 3.1.3.3

asynchronous 3.3.7.2

asynchronous combination 3.3.7.1

complete 3.3.8

component 3.3.5

component language 3.3.5

component relation 3.3.5

computation 3.1.1, 3.3.1

computation space 3.3.1

computation step 3.1.1

consistency 3.3.2

containing abstraction 3.3.4

discrete process 3.1.1

effective 3.3.3

effective procedure 3.1.3.3

embedded abstraction 3.3.4

formal 3.3.1

formal language 3.1.3.2

heuristic procedure 3.1.3.3

hierarchy of modularization 3.3.5

homogeneous 3.3.4

homogeneous development process 3.1.3.3

homogeneous methodology 3.1.3.3

informal attribute relation 3.3.6

informal attribute set 3.3.6

informal attribute set language 3.3.6

informally extensible 3.3.6

interpreter 3.3.1

methodology 3.1.3.3

modular 3.3.5

phase of a development process 3.1.2

process 3.4.1

process step 3.1.1

specification 3.3.1

specification language 3.1.3.2, 3.3.1

state 3.3.1

system 3.3.1

system space 3.3.1

system specification 3.1.3.2