THE FORMAL DESIGN AND ANALYSIS OF DISTRIBUTED DATA-PROCESSING SYSTEMS

by

D. R. Fitzwater

Computer Sciences Technical Report #361

August 1979

The Formal Design and Analysis of Distributed Data-Processing Systems

D. R. Fitzwater

ABSTRACT

This is the final report on ABMDSC-ATC-P Contract DASG60-76-C-0080, Mod. P00007 covering the period of May, 1978 to June, 1979. This work consists of the following major tasks:

a) To define a required set of formal properties for any distributed system specification language.

b) To analyze the Requirements Engineering and Validation System (REVS) for these properties.

c) To find a way to extend REVS for these properties.

d) To participate in the ongoing design experiments on distributed real-time systems.

The results of this work make possible the extension of REVS to the practical specification of distributed real-time systems and the automatic generation of simulation models from the specifications. In addition, new modes of distributed simulation of distributed systems are developed. A classification of real-time systems based on test simplification is also proposed.

TABLE OF CONTENTS

1. EXECUTIVE OVERVIEW

This section is intended as a brief summarization of this report and a listing of the major conclusions. This work was done under the continuation of the ABMDSC-ATC-P Contract No. DASG60-76-C-0080, Mod. P00007 during the period from 8 May 1978 to 30 June 1979. This report is the final report on the work of this contract.

1.1 The Problem

The RSL/REVS requirements methodology was developed to support specification of large-scale real-time control system requirements in the context of large centralized computer systems. Since that development there has been a growing realization that distributed computing systems offer some important possibilities as well as some new and not well understood complications. The extension of the previous centralized development methodologies to distributed systems is not simple and many techniques do not work at all.

Our previous work on a formalization of distributed system development led to a number of important properties that a distributed system specification should have in order to simplify the development problems. These properties also support significant extensions to current development tools.

This current work is an attempt to modify the RSL/REVS methodology to satisfy these required properties and to support the new tools for distributed system design.

The Advanced Technology Center has an ongoing set of experiments in distributed design that provide an opportunity to apply and test the proposed REVS modifications. We will use those experiments to validate this work to the extent possible in the contract period.

1.2 The Approach

The research plan is presented in detail in section 2. The work was factored into the following tasks:

- RR-0 Research Plan (CLN 6),
- RR-1 Formal Properties (SOW 3.1),
- RR-2 RSL Analysis (SOW 3.2),
- RR-3 MRSL Design (SOW 3.3),
  - RR-3.1 MRSL Semantics,
  - RR-3.2 MRSL Implementation,
  - RR-3.3 MRSL V & V Examples,
- RR-4 DDP Experiments (SOW 3.4),
  - RR-4.1 Develop DDP Example,
  - RR-4.2 Analyze DDP Example,

The planned approach was followed closely. The only major departures were in the implementations. The time and level of effort did not permit the actual modifications and extensions of the production RSL/REVS system to be done. Their nature and feasibility were instead demonstrated and validated by worked out prototype examples actually run on REVS.

1.3 The Results

The results of carrying out this research plan are summarized as follows:

RR-0 Research Plan (CLN 6),

The research plan is given in section 2, and was quite successful. A way to extend RSL/REVS for distributed system specifications with the required formal properties was found and demonstrated.

RR-1 Formal Properties (SOW 3.1),

The required formal properties were developed and defined in section 3. Their extension to certain types of performance attributes led to the design of some new models of distributed simulators/emulators that are capable of highly parallel execution, with resulting efficiency for distributed systems. A prototype design and implementation of one of these, an "equi-phase" multi-tasking simulator/emulator, was developed and

tested with a non-trivial example system specification. A classification of real-time systems based on simulation and testing efficiency was developed. This work was incomplete, but indicates a way to simplify the testing of real-time designs, while improving the reliability of the results.

RR-2 RSL Analysis (SOW 3.2),

The RSL/REVS system was analyzed for the formal properties as described in section 4. Since it was designed with other goals in mind, it is not surprising that most of the properties did not hold for it.

RR-3 MRSL Design (SOW 3.3),

The results of this work are given in section 5. The required modifications to RSL/REVS in order to specify and simulate asynchronously interacting processes are minor. The major additions are a standard run-time simulation package to support a re-interpretation of the interface concepts of RSL, and a new source language translator to the old RSL form. A prototype example of a distributed system specification in MRSL and the run time simulation package were developed and run on REVS. The source language translator is described in terms of phrase mappings into RSL equivalents.

RR-4 DDP Experiments (SOW 3.4),

The CS-1 DDP experiment was selected and a DPR for it in terms of asynchronously interacting processes was developed. A sample system specification in the same form was hand translated to an equi-phase multi-tasking program and simulated on the equi-phase simulator/ emulator. Some of the CS-1 DPR processes were analyzed (by hand) in terms of the equivalent MRSL type R-Nets. These results are given in section 6.

1.4 The Recommendations

The results of this work can be summarized in the recommendations of this section.

1.4.1 RSL Modifications

If RSL/REVS is to be extended to the specification and analysis of distributed systems, it should be carried out by designing and implementing the following:

a. An AIP-type source language.

b. A translator to equivalent R-Net form.

c. Standard simulation packages for modeling the modified concept of "interface" in RSL. There would be a package for each mode of simulation/emulation.

d. Additional formal analysis tools for the design data base.

The chacterization and prototype design of the above components is given in this report.

1.4.2 Distributed Simulation

The work reported in section 3 opens the way for the generalization of the conventional concepts (linearized event calendar) of discrete simulation to truly distributed simulator/emulators of asynchronously interacting processes. Such distributed simulators can be much more efficient in studying distributed systems than are the conventional ones.

The development of a hierarchy of process simulation/emulation models suitable for use at the DPR level should be continued to provide more analysis tools for a designer than the conventional simulator.

1.4.3 Real-Time Design

The theoretical work in section 3 may also be extended to provide formal (and testable) sufficient properties for real-time systems that will greatly simplify their testing and allow a designer to avoid unnecessary complications. The presence of such properties will also vastly improve the efficiency of simulations.

Theorems based on this work could be used to ensure that the many and subtle errors frequently introduced into designs and simulations due to the parallelism would be avoided.

We believe that further work in this area would be quite fruitful and that it should be pursued vigorously. The potential benefits in this very troublesome area are enormous.

1.4.4 Designer's Workbench

Long experience with the development of programs has led to the design of many tools to assist in producing and checking programs. Their collection and integration into development systems such as "the programmer's workbench" incorporated in the UNIX operating system can greatly facilitate programmer productivity. System designers, on the other hand, have had to do without such syntactic and semantic tools, and make do only with data base management tools for documentation and report generation. Even the simulation models for the specified systems were independently programmed with little assurance of consistency with the source specification.

The formal specification methodology developed in this and the preceding reports makes it possible to create (analogously) an integrated "system designer's workbench" that can be used to support distributed real-time system development.

The design of large-scale distributed systems is also a distributed process, and these same techniques can be used in the workbench design for implementation on networks. Because of the specialized nature of this workbench, it can be implemented even on small computers.

The development of a designer's workbench is a direct attack on the life cycle costs and unreliability of large-scale real-time distributed system development processes.

The required investments are relatively trivial in obtaining a working prototype that could be used in real design experiments. Such a system would also provide a common basis for evolutionary development of new tools and methodology based on design experience.

The potential payoffs are enormous and this approach deserves a trial. We have completed all of the theoretical work, and have started to build a "bootstrapping" prototype using the UNIX facilities. We believe that this work should be completed and tested.

2. Introduction

2.1 Background

This document represents the Final Report for the period beginning on 8 May 1978 of the continuation of BMDSC-ATC Contract No. DASG60-76-C-0080, Mod. P00007. A detailed outline of the research plan and the current status of the research is given in section 2. The main effort falls into two broad areas as detailed in sections 3-5. The first is to define a set of formal properties applicable to any systems specification language (section 3) and to analyze the Requirements Engineering and Validation System (REVS) software, especially the Requirements Statement Language (RSL) for these properties (section 4). The second is to find ways to extend or modify RSL/REVS so that the formal properties hold true for the extended or modified language (section 5). Since so much of the plan is devoted to a study of REVS software we provide an overview of the organization and intended use of that system.

2.1.1 Requirements Specification Language (RSL)

RSL is a language for entering, modifying, or deleting information in a data base, called the Abstract System Semantic Model, or ASSM (see [MD]). Information consists of

---

[MD] Dyer, Margaret E., et al. REVS Users Manual (SREP Final Report Volume II). TRW Defense and Space Systems Group, August, 1977.

"elements," "attributes of elements," and "relationships" between elements, where the quoted words are reserved in RSL, each corresponding to a set of types. For example, "data" and "contains" are individual types of element and relationship, respectively. (The meanings of these words generally conform to ordinary usage, as with "data" and "contains.") The element type "R-Net" is essentially a simulatable control structure constructed from other element types, notably "alphas," each of which has a Pascal procedure executable during R-Net simulation. However, some other element types as well as attribute and relationship types are for the purposes of clarifying, documenting, cross-referencing, and testing specifications and do not affect the simulation of R-Nets at all. Examples include "originating_requirement," "documents" (a relationship type), and "description" (an attribute type).

R-Nets have formally conventionalized pictorial representations as exemplified by Figure 2-1. (Note that alphas are denoted by rectangles in the figure.) R-Nets may receive information as messages through input interfaces from drivers (Pascal procedures defined externally to RSL, but necessary for the simulation of R-Nets); likewise, they may pass information through output interfaces to drivers. Both types of interfaces are denoted by hexagons in Figure 2-1 with input interfaces at the top and output interfaces at the bottom of R-Nets. Flow of control is illustrated by arrows

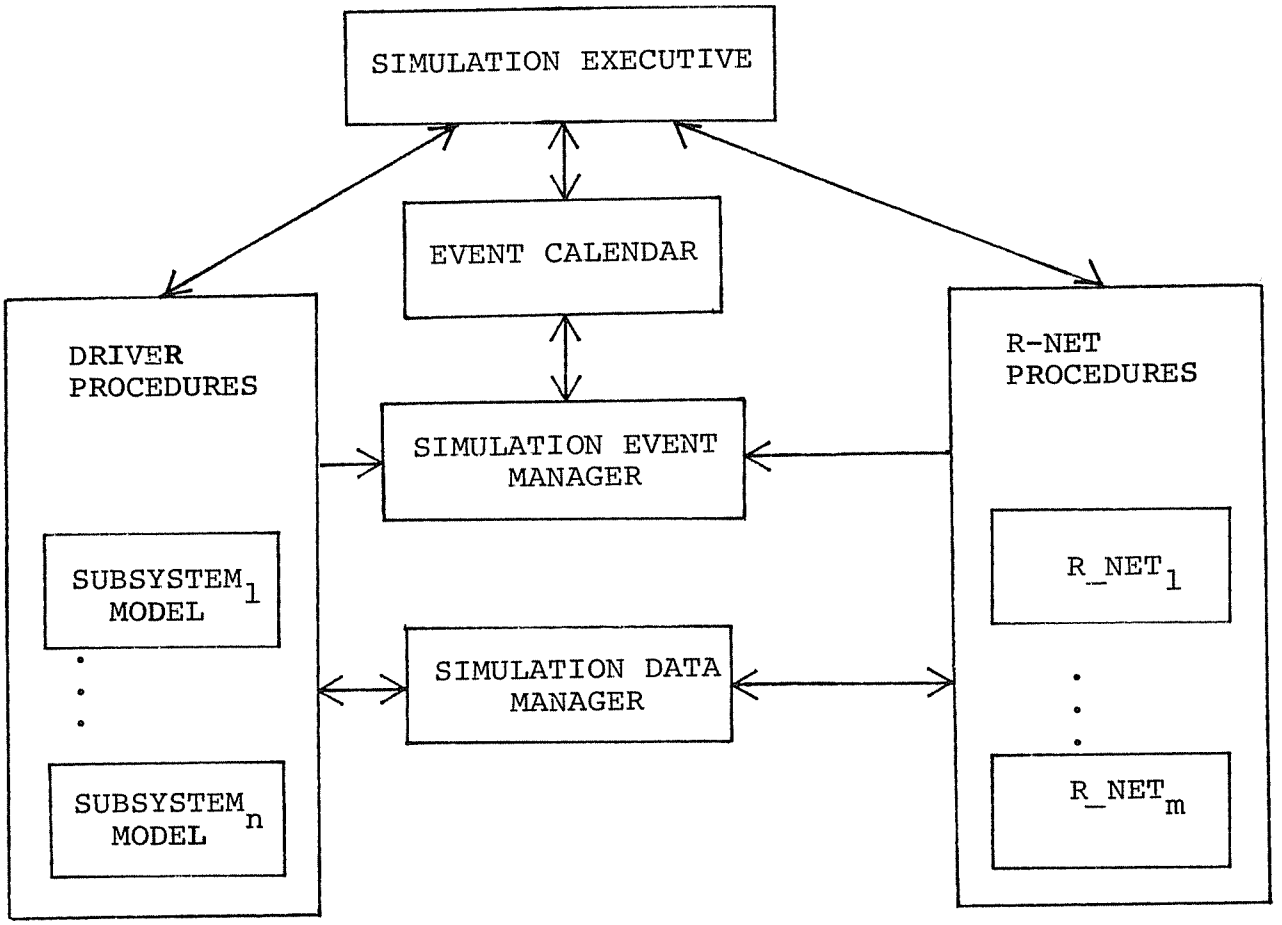


Figure 2-1. Components in Simulator Produced by SIMGEN (Fig. 7-1 of [MD])

connecting nodes. Messages passing through interfaces as well as the data and files which constitute messages are declared in RSL, of course. RSL is also used to define the input to alphas and the output from them. For a more detailed explanation of RSL see [MD], section 3.

2.1.2 Requirements Engineering and Validation System (REVS)

As we have already mentioned, R-Nets as defined in RSL cannot be simulated in isolation by REVS, since drivers (Pascal routines provided externally to RSL) are necessary to receive messages from the R-Nets and to generate messages for them. (Drivers are intended to simulate events in the real external world.) Passing of a message to a driver or to an R-Net causes the REVS simulation executive to schedule an activation of the respective driver or R-Net on the event calendar (see Figure 1-2). The simulation data manager is responsible, among other things, for the actual passing of messages back and forth between the driver procedures on the left side and the R-Net procedures on the right side. The four rectangles representing the simulation routines in the middle are provided by REVS and are influenced only indirectly by R-Nets and drivers through special language constructs and predefined procedures. For example, events cannot be examined or removed from the event calendar by user code. In other words then most of the operation of the REVS simulation software is transparent to the user.

Some facts about simulation time in REVS are pertinent

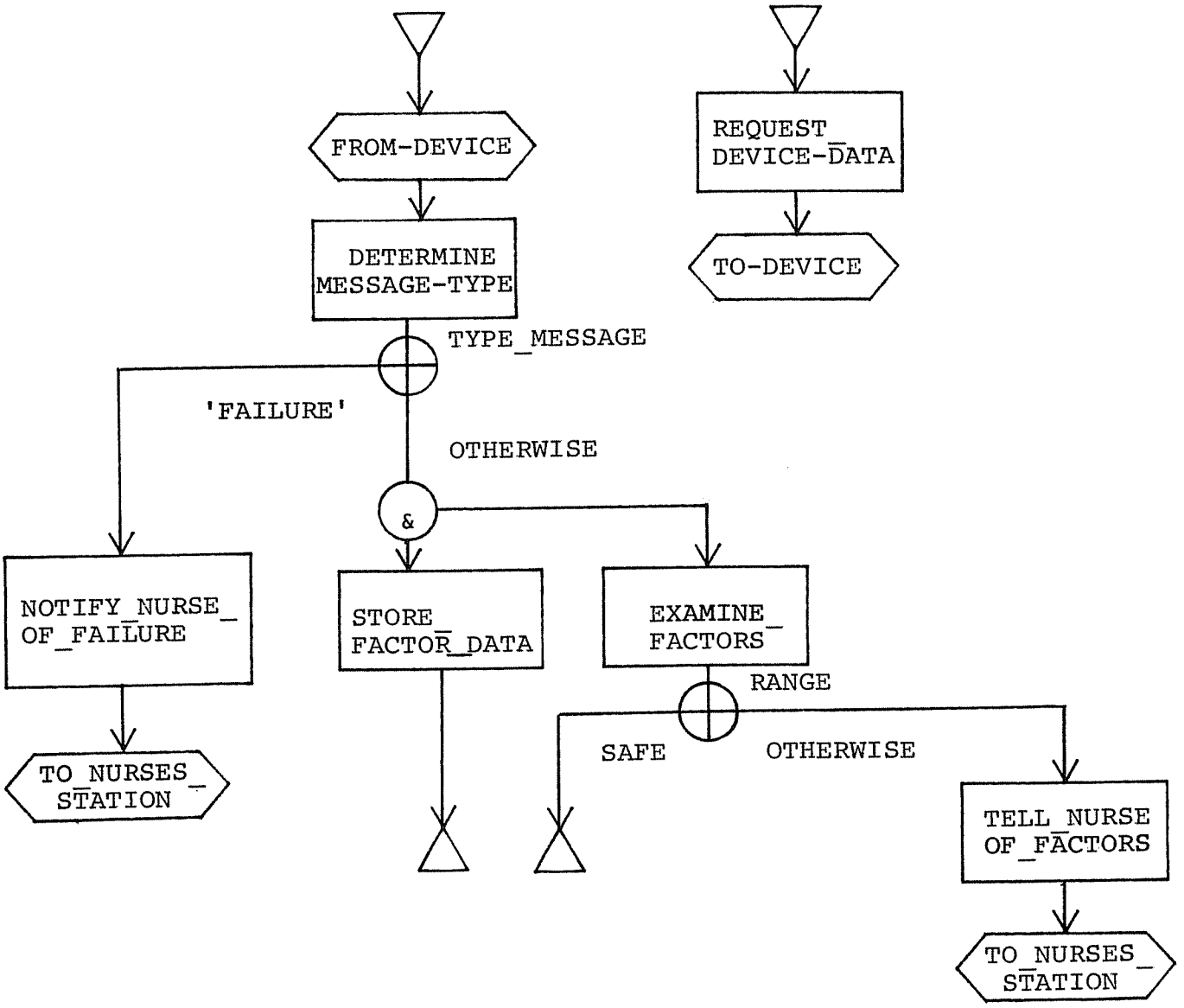


Figure 2-2. Example R-Nets (Fig. 6 of [A1]

[A1] Alford, Mack W. "A Requirements Engineering Methodology for Real-Time Processing Requirements." Trans. Soft Eng. SE-3, January 1977.

to much of our later discussion. For example, in terms of simulation time R-Nets and drivers are evaluated instantaneously. However, R-Nets may activate other R-Nets with a delay in simulation time, yet R-Nets activated in such a way can never receive messages but only send them. Note that an R-Net can schedule an arbitrary number of drivers (one per message output), but drivers, once activated, can schedule R-Nets at any arbitrary time afterwards. They can also schedule driver events by means of a special user-defined exogenous event routine. In section 5 we outline a new interpretation of message passing in RSL/REVS consistent with our definitions of asynchronously interacting processes.

We will now summarize by briefly mentioning the software packages that can be called within REVS via its own executive language. These are illustrated in Figure 2-3 in the approximate order in which they would typically be used in developing and testing a set of specifications and requirements. First one builds a data base with the RSL package and then analyzes it for inconsistencies, omissions, and other errors with the Requirements Analysis and Data Extraction (RADX) package. RSL and RADX may be used jointly any number of times until the desired data base is achieved. After drivers and their related procedures and declarations are written one can then call the Simulation Generation (SIMGEN) function of REVS. The SIMGEN function generates a Pascal simulation by inserting user code, appropriately transformed as necessary,

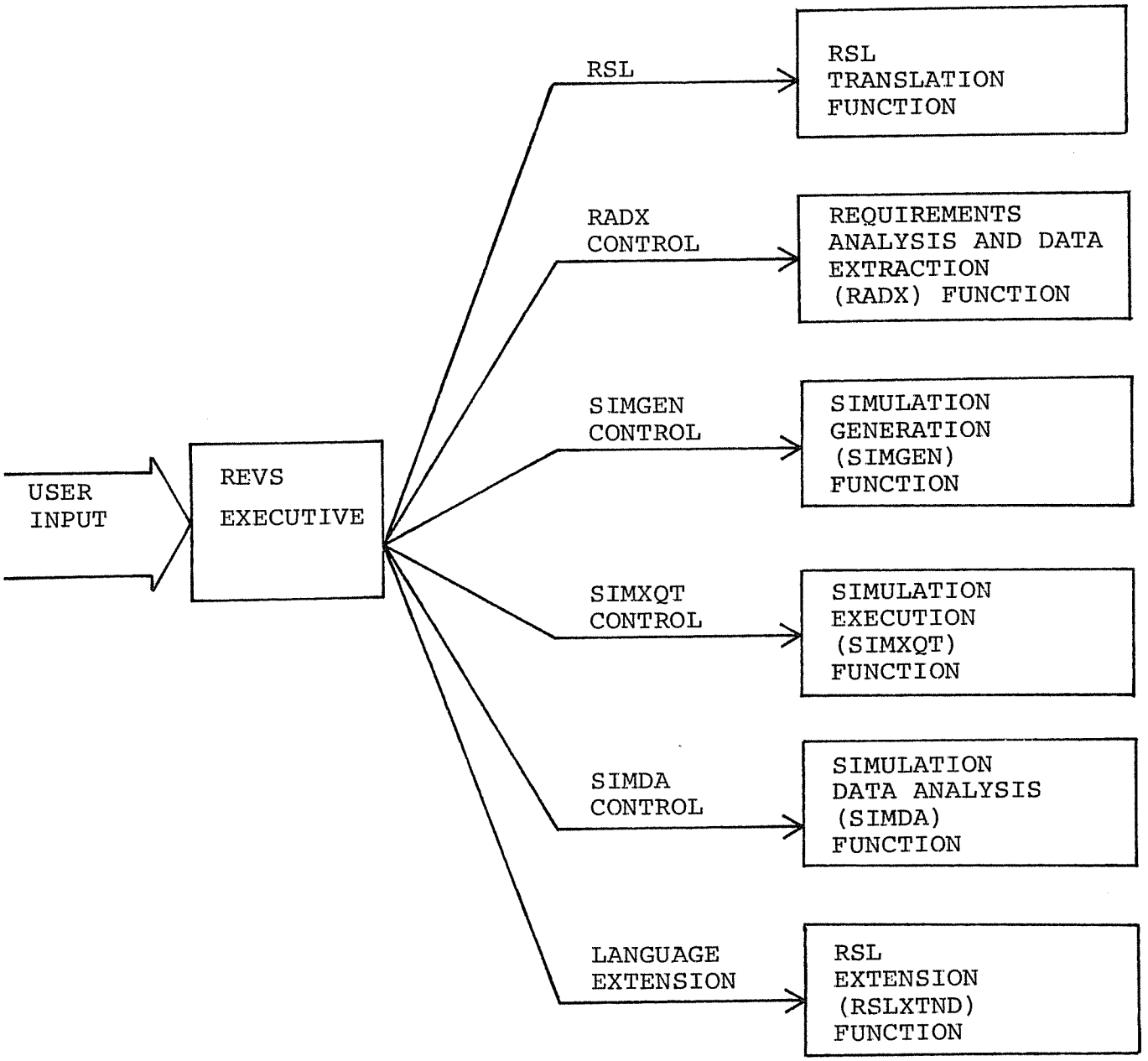


Figure 2-3. Functions Available to REVS Users
(Figure 4-1 of [MD])

into a host program. Errors may, of course, be detected at this point, especially during compilation of the resulting Pascal program. When errors have been removed to the designer's satisfaction he may proceed with actual simulation. Upon invocation of the Simulation Execution (SIMXQT) function of REVS the simulation program is executed and output data from the program is stored for use by a subsequent analysis function. This analysis is performed by the Simulation Data Analysis (SIMDA) function of REVS. All of the REVS functions may be used again on any subsequent specifications as the designer produces more detailed or complete approximations to his desired end product. For a much more thorough description of REVS software see [MD].

2.2 Research Plan

The following outline was the basis for the major portion of a presentation on August 3rd for the Distributed Data Processing Review conference held at Treasure Island, Florida.

2.2.1 RR-0 Research Plan (CLN 6)

2.2.1.1 Requirement

The Contractor shall submit a program plan for meeting the research requirements added by this modification. The said program plan shall be submitted to the Government within two (2) months from the effective data hereof.

2.2.1.2 Current Status

The research plan was completed and presented to the Government July 10 and 11, 1978. Section 2.2 contains a brief discussion of the resulting plan.

2.2.2 RR-1 Formal Properties (SOW 3.1)

2.2.2.1 Requirements

The Contractor shall define and document the first-order set of specification properties that provide for the precise specification of DDP systems. These properties shall be consistent with the results from previous research but extended as required through findings during this contract period.

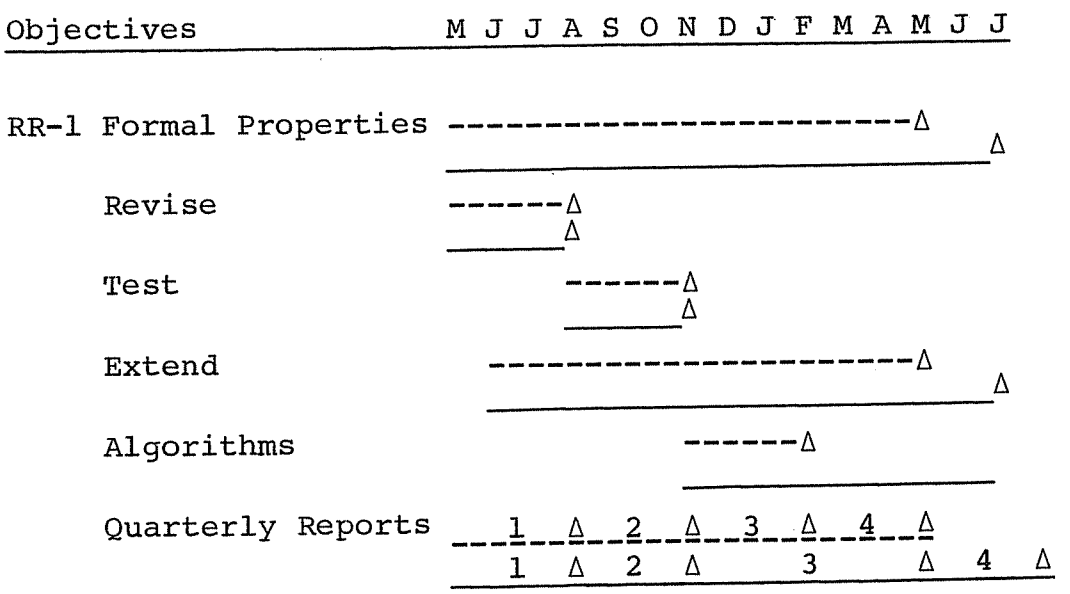
2.2.2.2 Objectives

* Revise formal specification properties to improve their usefulness in supporting development methodologies and to simplify their application
* Extend the formal properties to include practical performance attributes required for dynamic analysis
* Develop algorithms for testing properties in the modified RSL/REVS system

2.2.2.3 Current Status

The formal properties were revised and extended to support more efficient distributed simulation and testing of realtime systems. The results are in section 3.

2.2.2.4 Schedule



Projected times are shown by dashed lines.
Resulting times are shown by solid lines.

2.2.3 RR-2 RSL Analysis (SOW 3.2)

2.2.3.1 Requirements

The Contractor shall conduct analysis of the BMD Requirement Statement Language (RSL) to determine which formal specification properties that it does not contain. This will include the explicit specification property for specifying asynchronous processes and their interactions. The Contractor shall provide a list of the formal specification properties not contained in RSL and a recommended plan for the incorporation of these properties into RSL.

2.2.3.2 RR-2.1 RSL Properties

1. Objectives

   * Develop formal models of RSL/REVS
   * Analyze models for specified properties
   * Identify major deficiencies

2. Current Status

   A number of formal RSL/REVS models has been developed and analyzed. RSL/REVS is found to be seriously deficient in most of the specified properties. The results of this task are documented in Section 4.

2.2.3.3 RR-2.2 Modified RSL (MRSL) Approach

1. Objectives
   * Identify how RSL/REVS could have the specified properties
   * Develop a research approach to give RSL/REVS the specified properties
2. Current Status
   * We have studied the following approaches:
     A. Augment RSL/REVS
     B. Specialize RSL/REVS
     C. Constrained generation of RSL/REVS
   * We have developed a research approach:
     A. Identify RSL/REVS specializations (critical issue: how to map asynchronous processes?)
     B. Design enforcement modifications (critical issue: generate or test?)
     C. Implement designed modifications (critical issue: integration into RSL/REVS)
   * Task RR-3 is designed to carry out this research approach.
   * The results of this task are described in section 4.

2.2.3.4 RR-2.3 MRSL Properties

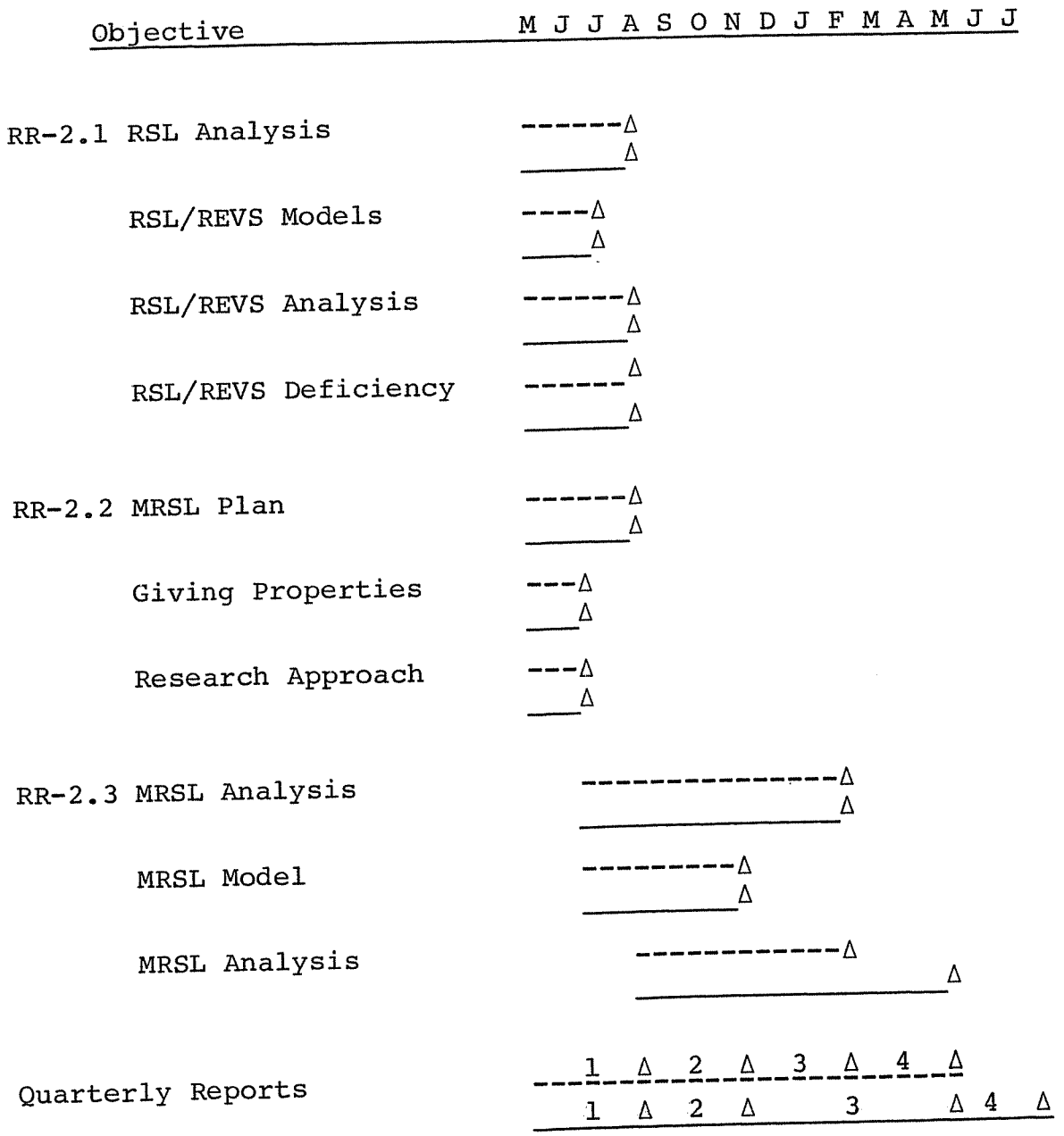
1. Objectives
   * Define specified properties in MRSL terms

* Show that MRSL has specified properties to a useful extent

2. Current Status

The MRSL specifications can only be shown (practically) to have the formal properties while in a more abstract functional form than an RSL specifications. The definitions and testing must be done on a suitable source language specification that can be translated into MRSL terms. The translation is complex but practical. The results of this task are shown in section 4.

2.2.3.5 Schedule



Projected times are shown by dashed lines.

Resulting times are shown by solid lines.

2.2.4 RR-3 MRSL Design (SOW3.3)

2.2.4.1 Requirement

The Contractor shall establish the design extensions to RSL for the incorporation of those formal specification properties that were recommended to be incorporated in RSL during this contract period. These extensions shall include formal analysis concepts, as well as, the formal specification properties. The Contractor shall provide assistance in the implementation of the design extensions to RSL. This shall include the validation and verification of the implementation.

2.2.4.2 RR-3.1 MRSL Semantics

1. Objectives
   * Interpret R-Nets as processes
   * Interpret interfaces as interactions
   * Develop specialized drivers
   * Enforce RSL subset constraints
2. Current status

   A detailed study of MRSL semantics is given in section 5. The enforcement of constraints must be carried out prior to MRSL translation.

2.2.4.3 RR-3.2 MRSL Validation and Test Examples (SOW3.3)

1. Objectives
   * Develop MRSL test plan
   * Develop MRSL examples

* Carry out test plan

2. Current status

The MRSL test plan was specialized to the development of a system specification example that would test, demonstrate and prove the validity of the MRSL concepts. This example is described in section 5.

2.2.4.4 RR-3.3 MRSL Implementation (SOW3.3)

1. Objectives

* Implement special drivers

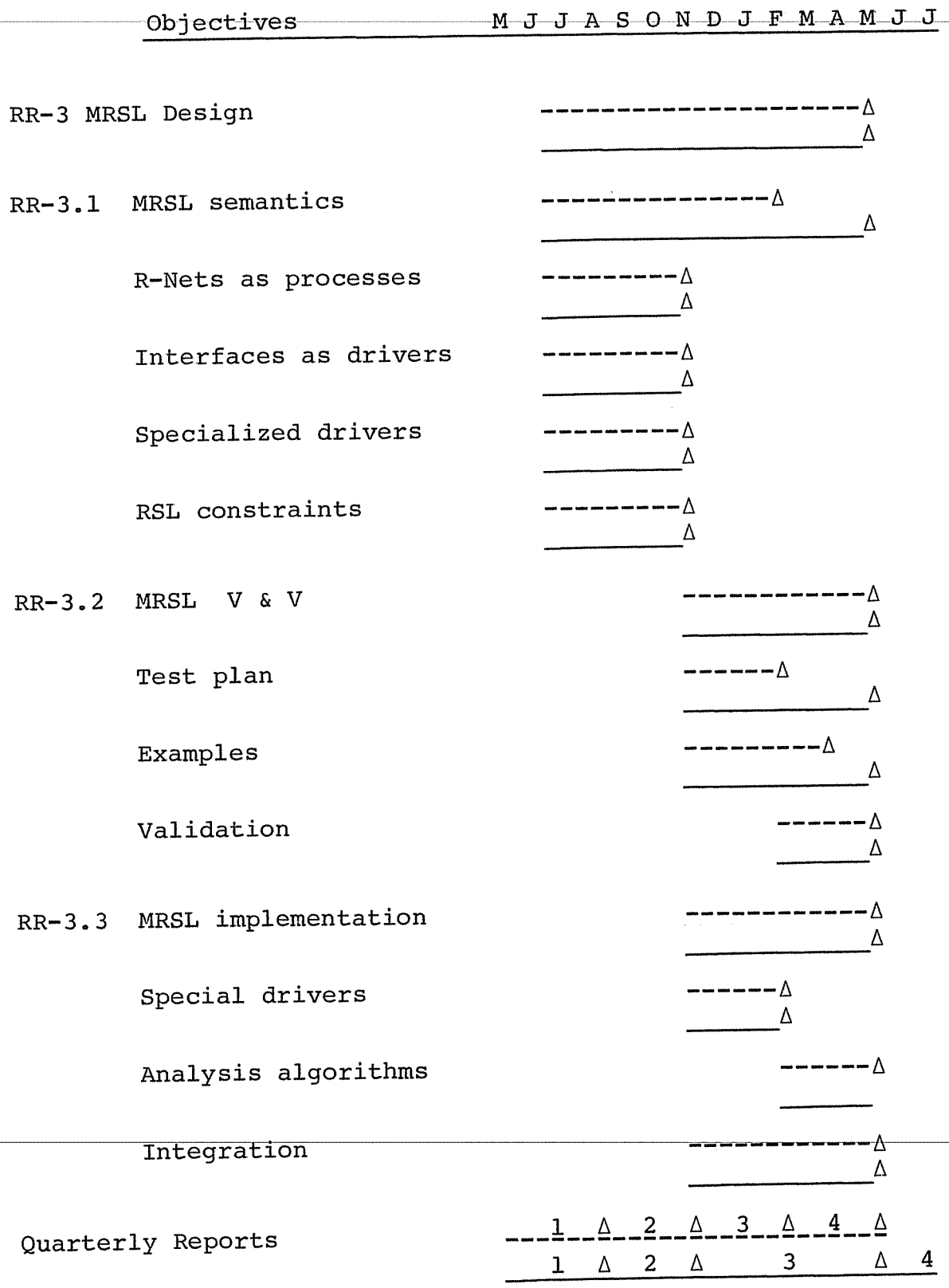
* Implement property analysis algorithms

* Integrate with REVS

2. Current status

The special drivers and REVS integration were developed and tested by actual running of the example developed in RR-3.2. The results are described in section 5.

2.2.4.5 Schedule



Projected times are shown by dashed lines.
Resulting times are shown by solid lines.

2.2.5 RR-4 DDP Experiment

2.2.5.1 Requirement

The Contractor shall prepare a formal specification for the BMD experiment to be designed by either the distributed processing requirements contractor (GRC) or the distributed processing architecture design contractor (TRW). The specification is to be written using the extended RSL specification language and the Contractor shall demonstrate the effectiveness of the language modifications. Additionally, the Contractor shall show the usefulness of the extended language in the analysis of the specified process.

2.2.5.2 RR-4.1 Develop DDP Example

1. Objectives
   * Select DDP experiment specification
   * Translate to MRSL equivalent
2. Current status

   The selected DDP experiment was Case Study 1 (Revised) of the BMDSC-ATC-P integrated case studied.

   The translation of CS-1 requirement specification into MRSL form by hand was impractical. Instead, a CS-1 DPR in a more abstract (and testable form) was produced. This specification is discussed in section 6.

2.2.5.3 RR-4.2 Analyze DDP example

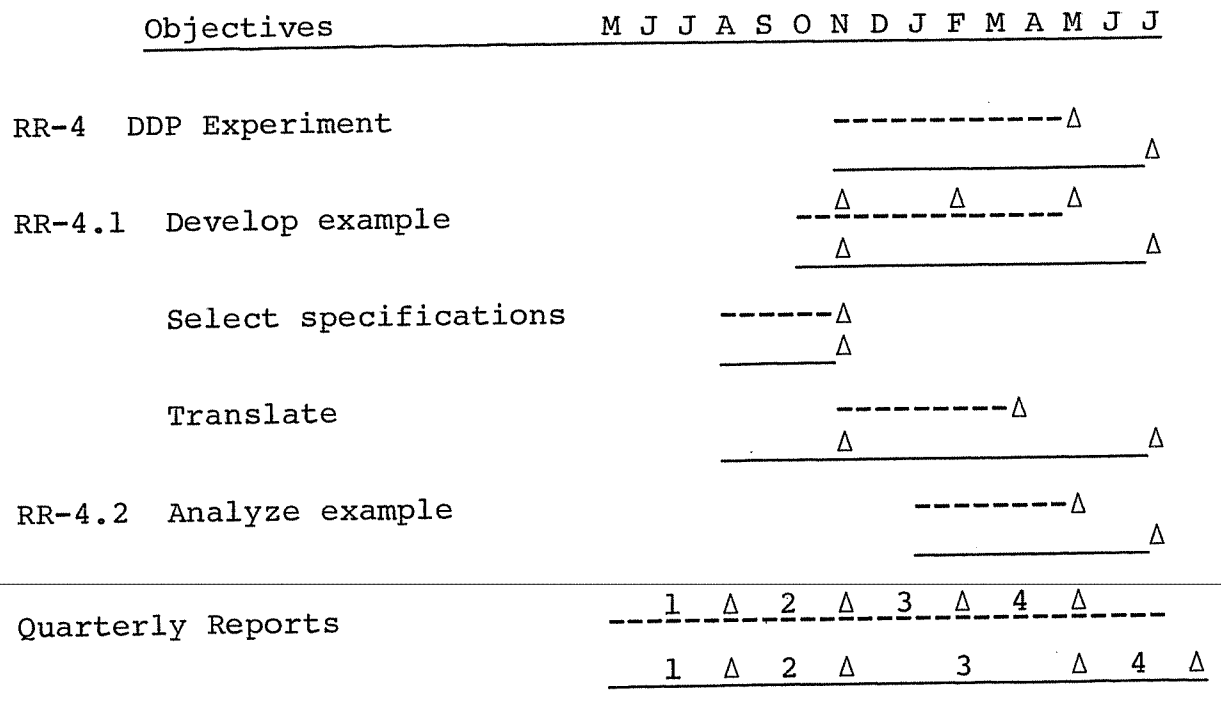
1. Objectives

   * To demonstrate MRSL capabilities
   * To analyze BMDSC-ATC DDP experimental specifications

2. Current Status

   The MRSL capabilities were demonstrated by the example in section 5. The CS-1 DPR was partially analyzed by translating a portion of it into a multi-tasking distributed simulation that was run on a computer. The results are shown in section 6.

2.2.5.4 Schedule



Projected times are shown by dashed lines.

Resulting times are shown by solid lines.

2.2.6 Summary

2.2.6.1 Current Status

RR-0 (CLN 6) Research plan: completed.

RR-1 (SOW 3.1) Formal Properties: revised and extended.

RR-2 (SOW 3.2) RSL analysis

- RR-2.1 RSL analysis: completed.
- RR-2.2 MRSL plan: completed.
- RR-2.3 MRSL analysis: completed.

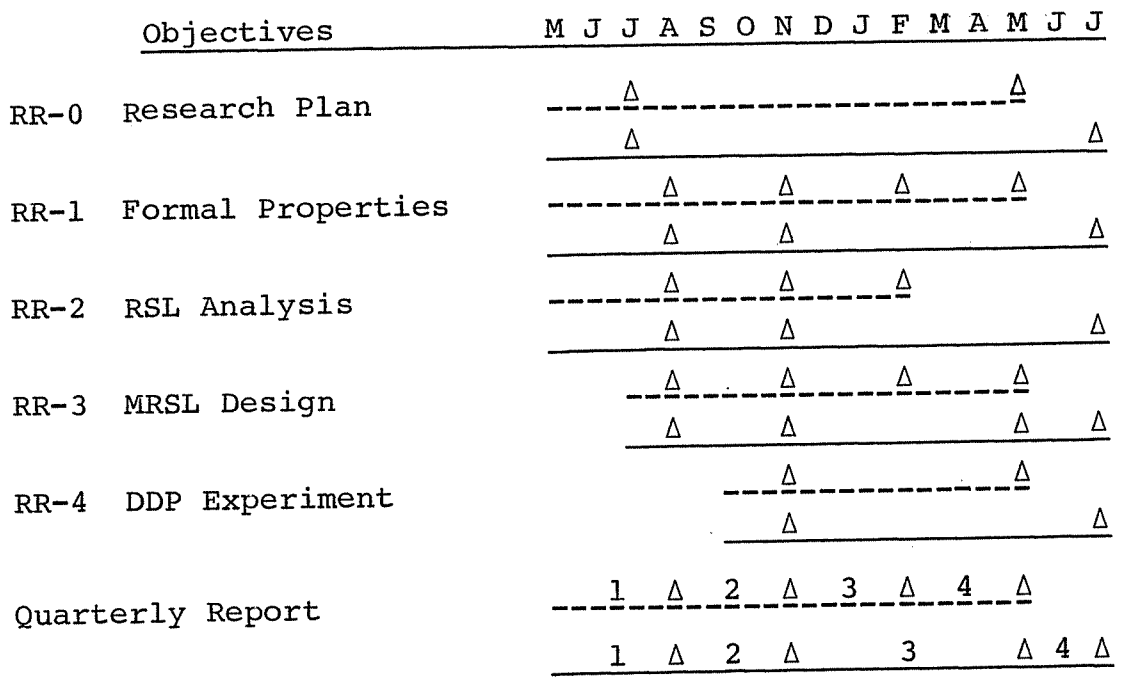
RR-3 (SOW 3.3) MRSL design

- RR-3.1 MRSL semantics: completed
- RR-3.2 MRSL implementation: prototype done.
- RR-3.3 MRSL V & V examples: completed

RR-4 (SOW 3.4) DDP experiment

- RR-4.1 Develop DDP example: CS-1 DPR done.
- RR-4.2 Analyze DDP example: initiated.

2.2.6.2 Schedule



Projected times are shown by dashed lines.

Resulting times are shown by solid lines.

3. Formal Properties (RR-1)

3.1 Initial Properties

The following definitions of specification properties are a starting point for the development of a formal development methodology. Not all specifications in any practical language will have all these properties. We do require that a specification be testable in a practical way for them. A formal methodology based on these properties is described in [Fz], [Fi78], and [Fi322]. We briefly present here the revised set of formal properties used in this study.

3.1.1 Systems

We must first define the concept of a system as a set of computations.

Let B be a finite alphabet

A system state is a finite string over B.

A computation is a countable sequence of system states.

A computation space is the set of all computations.

[Fz] Fitzwater, D. R., and P. Zave. "The Use of Formal Asynchronous Process Specifications in a System Development Process." Texas Conference on Computing Systems, November 1977.

[Fi78] Fitzwater, D. R. "A Decomposition of the Complexity of System Development Processes." IEEE Computer Society's Second International Computer Software and Applications Conference, November, 1978.

[Fi322] Fitzwater, D. R. "The Formal Design and Analysis of Distributed Data-Processing Systems," CSTR 322, University of Wisconsin-Madison, April 1978.

A system is a non-empty set of computations such that each state has a finite number of possible successor states, and the future course of a computation is dependent only on its current state.

An asynchronous combination of a set of systems is a system in which computations are an interlacing of computations of the component systems.

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

If a system has asynchronous interactions between its component systems, the asynchronous combination of those components will be a containing abstraction of the interacting combination of those components.

3.1.2 Specification Properties

At least the following formal properties of system specifications are required as a basis for a formal methodology.

Let A be a finite alphabet.

A specification is a finite string over A.

A specification language L is the set of all specifications.

A system space S is the set of all systems.

An interpreter of L in terms of S is a function

$I : L \rightarrow S$.

A simulator of $\ell \in L$ is an algorithm, $(A(\ell,a)$, which given any state "a" appearing in a computation in $I(\ell)$, will produce the finite set of all immediate sucessor states.

A component language K is a set of finite strings over A.

A component relation $R_K = \{(\ell,k) \leftarrow L \times K : k \text{ is substring of } \ell\}$, decomposes a specification into substrings.

An abstraction relation $R_A \subseteq K \times K$. If a specification ℓ has a component k_2 and $(k_1,k_2) \in R_A$, we may substitute k_1, for k_2 in ℓ_2 to obtain a specification ℓ_1. If both ℓ_1 and ℓ_2 are consistent then $I(\ell_1)$ is an abstraction of $I(\ell_2)$.

Formal

In order to have a formal specification system, in which the required properties are meaningful, we must have the definitions of the following:

A, B	finite alphabets
L	a specification language
I	an interpreting function
K	a component language
R_k	a component relation
R_a	an abstraction relation

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 $\ell \in L$ is consistent iff $I(\ell) \in S$.

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

Effective

A specification $\ell \in L$ is effective iff we have a simulator of ℓ.

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 or the specified system.

Modular

A specification $\ell \in L$ is modular iff ℓ is consistent, and there is a substitution for components of ℓ defined by R_A.

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

Homogeneous

A specification $\ell \in L$ is homogeneous iff ℓ is modular and substitutions defined by R_A lead to consistent specifications.

Every abstraction of a system must also have a formal specification. The same specification language may be used

throughout the development process.

Asynchronous

A specification $\ell \in L$ is asynchronous iff $I(\ell)$ is an asynchronous combination of interacting systems.

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.

A discussion of these properties in the context of RSL is given in section 4.

3.1.3 Processes

An initial "solution" to the above constraints can be formed out of asynchronously interacting processes. A process serves as a generator of computations of the system specified by that process. A specific functional interaction mode using exchange functions is also included.

A process state is a string over $V \subseteq A$.

A process state space is the set of all process states.

A state successor relation is a mapping from a process state space to a process state space.

An isolated process is a tuple (f, s) of successor function and state space.

A process specifies a system by defining a generator of its computations (f, s) specifies

$$S = \{c: \forall s_i \in s,\ c = \langle s_i, f(s_i), f^2(s_i), \ldots \}$$

Each function may have attributes of evaluation time interval and space.

Each non-primitive function and set will be defined as an expression of primitive functions and sets.

Each primitive function and set may be defined as an expression of still more primitive functions and sets.

Asynchronously Interacting Processes (AIP)

Let each process (f_i, s_i) specify an interacting system S_i.

The set of asynchronously interacting processes $((f_1, s_1), .., (f_n, s_n))$ specifies the interacting combination of the S_i.

The only effect of interactions is to eliminate some of the computations in the asynchronous combination of the systems S_i.

We can study isolated AIP confident that they will generate no new behavior when combined with others.

Asynchronous interactions require

* Information exchange between "functions" as a side effect of evaluation
* Evaluation synchronization of one does not complete before interacting other initiates
* A set of new primitive "functions" XC, XA, and XS

* XC: If $XC_i(A)$ and $XC_i(B)$ are the only two instances of XC in class i and an interaction takes place, we have $XC_i(A) = B$ and $XC_i(B) = A$.
* XA: An XA_i is treated the same as an XC_i except that XA_i will interact only with XC_i or XS_i.
* XS: An XS_i will interact only with XA_i or XC_i. If no XA_i or XC_i is pending when $XS_i(A)$ is evaluated, it does not wait and completes by taking A as its value.

The major goal of this research approach is to find a way to map this form of solution into RSL/REVS structures with minimal perturbations of the existing methodology.

3.2 Performance Attributes

The large and informal set of all possible kinds of performance attributes may never be completely formalized. We can extend our formal properties to include some kinds of performance. The primary constraint on our extension is that we must be able to prove automatically that the formal performance specification is consistent using just the formal specification itself.

First we will note that, for any system specification, we may have at least three classes of performance specifications

(1) of the system computations,

(2) of the system realization, and

(3) of the development process.

Clearly, we can formalize, as a part of a system specification, only the first class of requirements, since it is only the system itself that is formally specified. We may at times wish to study the realization system or the development system by formally specifying them in the same fashion. The other classes of performance attributes can then be treated in the same way we will treat the first class.

3.2.1 Validation

We can also classify performance requirements by the way in which they can be verified and validated (V & V) on a specification. The major ways are informal, observational, and theoretical. Of course, an arbitrary performance specification may require the use of all three types of V & V.

Informally validated performance requirements are, by definition, beyond the scope of our formal treatment unless the V & V can be somehow formalized. Any acceptable technique could be used.

Observational validation implies the actual generation of computations of a specified system in experiments designed to test the performance requirement.

Theoretical validation implies the analysis of a system specification itself to show the validity of a performance requirement.

A performance requirement may be only informally testable at some stages of the development process, and be

formally validated at a latter stage. For example, the amount of performance degradation introduced by resource contention in a system realization may not be well defined until the realization system itself is developed.

3.2.2 Primitive Attributes

The simplest (and still quite useful) extension to our formal properties is the introduction of a formal attribute of time for each primitive function and of space for each primitive set in a specification.

Establishing a value for such attributes is not trivial. The time required in a realization of a function may be dependent on the actual argument values, data access conflicts with other tasks, control conflicts among processors, and resource allocation conflicts. Thus, to assert precisely the time it would take to evaluate a function, one must formally include aspects of the realization of the system that have not, as yet, been specified. Such a precise assertion would be impractical. How then can we guarantee consistency in such performance allocations? It is easy enough to just assign arbitrary values, but how can we know that such a system is possible even in principle (no realization technology constraints)?

A simple way to answer these severe questions is to use independent random variables as attribute values. If the variable reflects an expected distribution of argument values and resource conflicts it can be considered as a derived value from some other performance requirements. If

not, then the random variable can at least reflect expected distributions of "infinite" (or unbounded) resource models of realization. It then becomes an optimistic (any realization might degrade that performance through resource contention) value. If even the optimistic values cannot meet the requirement then immediate redesign is required and many development resources are saved. Without the assumption of independence, any formal attribute consistency seems to be impractical to demonstrate.

Internal consistency is maintained since the time (or space) attribute of composite functions (or sets) can be derived from that of the associated primitives. The feasibility of a realization of such specifications can only be informally validated on that specification. A development process may, at times, call for a "feasibility breadboard" realization to establish such informal confidence in the performance allocations implicit in attribute value assignment.

A frequent type of performance specification is that of "port to port" times for response to stimuli. The validation of such a requirement then consists of an observation of such times in a simulated computation. The design of the particular simulation experiments required to display the information may be a problem, but the ability to carry them out is guaranteed by the formal specification properties. Under certain special conditions, the timing analysis could be carried out on the specification itself.

3.2.3 Factorization

The most difficult task is to factor a performance requirement into the components applicable to each of the classes in a way that is probably consistent. The resolution of this problem is beyond the scope of this report.

A possible approach would be to design in each phase so as to maximize the stability of the decisions to the perturbations of subsequent phases. For example, a design that is functionally correct for any execution speed will be stable to all subsequent allocation and optimization decisions, while freeing subsequent design choices. In this case, the performance requirements are factored into the subsequent phases. See section 3.3 for details of such real-time invariancies.

3.3 Distributed Simulation/Emulation

The extension of the formal properties to performance requirements requires us to specify simulation and emulation models that can demonstrate the behaviors of the specified system. There are two problems involved, the performance semantics, and the efficiency of the simulator/emulator. In both cases, we really need a distributed simulator for distributed systems.

3.3.1 Discrete Simulation

The normal discrete simulator is designed to run on a single processor system using a linear ordering of event routines to model parallel processes. This requires the

simulation programmer to make a number of semantically important assumptions to obtain a valid simulation. Usually, efficiency concerns require approximation compromises that may or may not be valid. These assumptions are not a part of either the specification or of the final simulation program and are not shown to be consistent with the specifications. For our purposes, this is not good enough. We want to specify asynchronous process performance formally and to validate it formally without dependence on informal (mis-) understandings as to their interpretations. For efficiency, simulators may (and emulators do) require distributed realizations while modeling distributed systems. The normal way to construct simulations makes it impossible to exploit a distributed realization, since all events have been linearly ordered and must be evaluated in that order. We must first develop models for distributed (rather than linearized) simulation.

3.3.2 Distributed Simulation

We want to machine translate our asynchronously interacting processes (AIP) into executable code that itself defines the simulation model for further experiments. The resulting programs must run validly on a distributed implementation that allows arbitrary allocation of processes to components of the implementation. This allocation may be essential for emulation as well as for improving the efficiency of the simulation.

We may have to introduce new formal properties for specifications in order to ease distributed realization. For now, we will assume only that the specification to be simulated is consistent.

3.3.2.1 Task Decomposition

The simulation of an AIP primarily consists of the parallel evaluation of arithmetic expressions. For simplicity and familiarity, it will be discussed in those terms.

The usual translation of expressions to sequential programs involves the assignment of the implicitly required temporary variables and of a sequential flow of control. For our purposes, we must be able to specify multiple interacting control flows, while preserving the integrety of the process state spaces.

For our distributed realization model (the target "machines" for our translator) we will assume that each "machine" consists of one or more processors and a single shared memory. The set of machines will communicate via a shared buffer memory. The details are irrelevant to this discussion.

The current state of each process will be allocated to a single machine, and the expression defining the state successor function will be translated into a set of interacting tasks. A task is a normally compiled program with a sequential flow of control and a set of inter-task

communication functions. Tasks may be evaluated in parallel. Each task may read the initial state values of its process and its argument values. After sequential execution of its associated program, the task will produce a value to be used as a new state component or as an argument to another task. When the successor process state value has been formed, it becomes the new initial process state value and the process state successor function is re-evaluated.

The translation to tasks is relatively straightforward at the current state of the art. The accessing rules above guarantee that critical access conflicts cannot occur and that variables remain well defined in spite of the parallel execution of tasks. With the exception of the interaction functions in the task program, the task translator is just that required for parallel evaluation of arithmetic expressions. This is a much simpler job than translating sequential programs (such as Fortran) into multi-processor form.

Each machine will have a queue of pending tasks. An idle processor on that machine will remove and execute the first task in the queue. Task execution may result in the creation of more pending tasks, in the creation of next process state components, or in the initiation of interactions with other tasks. In the latter case, the task will "sleep" until the completion of the interaction and the processor will be idled. In any case, all pending tasks may be executed at any time and in any order while preserving the validity of the simulation.

We have several options in the translation of primitive sets and functions for our simulation/emulation. A primitive set is effectively a primitive type, and any function argument whose value is a member of a primitive set can only be transformed by primitive functions whose domains are of that type.

The evaluation of a primitive function by the simulator will consume time and space resources as well as producing a value in its range. The resource consumption may be specified by formal attributes of the function (simulation) or by the actual consumption during evaluation (emulation).

The value of a primitive function may be obtained by evaluating one of the following:

(a) a stochastic function with the range of the primitive function it replaces,

(b) an implementing algorithm at a suitable level of approximation, or

(c) a standard implementation of an imtrinsic (built into the interpreter) primitive function.

Each of these choices represents different modes of simulation.

3.3.2.2 Timeless Simulation

The behaviors of a system specification without time attributes will generally be quite non-deterministic. Each process state might have many immediate successor states. A timeless simulation would then produce from a given system

state all possible successor states reachable at any evaluation speed. Each such successor state would correspond to a particular sequence of task interactions. This is the most general form of simulation, but combinatorial complexity may restrict its use to a few special cases.

We may simplify the simulation complexity by selecting a particular interaction sequence to follow, and allowing the simulator to continue from system state to system state. Ideally, we would select a member of a large equivalence class of behaviors and study that one in detail. For example, if the system were deterministic in all of its interactions, its behavior at any particular evaluation speed would be equivalent to that at any other speed.

An interesting simulation can be generated by choosing the interaction sequence as follows:

(a) run all tasks until they go to sleep awaiting interactions.

(b) when the task pending queues are empty (i.e. nothing more can be done until an interaction occurs), carry out the currently enabled interactions and awake the associated tasks. Go to (a).

The evaluation sequence selected in this way is only a function of the logical structure of the specification, and the resulting data can be directly related to that structure. We will call this an equi-phase simulation. A phase step consists of one pass through the above procedure, and all processes will progress at the same rate in terms of

phases. The resulting data is invariant to the distribution of the simulator, which may be designed for efficiency.

We have not completed a theoretical analysis of this simulation mode to determine the special conditions under which it is sufficient to display all system behavior. Such a theorem would vastly improve simulation efficiency for suitable distributed real time systems.

3.3.2.3 Timed Simulation

When time (and space) attributes are specified, the simulation can time stamp each interaction initiation and completion. Potentially non-deterministic interactions may be resolved using priority assigned to the oldest initiated interaction. So long as this principle is not violated, the equi-phase constraint on simulator operation can be relaxed to permit still greater simulation efficiency. The resulting data is again invariant to the distribution of the simulator, while allowing a maximum degree of parallelism in the simulation.

The theoretical analysis of this mode of simulation should lead to recognition of regions of stability (invariant behavior) to perturbations in the time and space attributes. The resulting decomposition of the complexity of real time behavior simulations will improve both the efficiency and the reliability of such testing.

3.3.2.4 Emulation

If the simulator uses the actual (simulator) expenditure

of time and space to control the interaction sequences, it becomes an emulator. This mode is possible even if the specification is timeless. The emulation is even more efficient than the timed simulation, since it never needs to wait in completing an interaction until simulated "now" catches up to simulator "now". The resulting data is, of course, not invariant to the distribution of the simulator, and studies of the behavioral stability must be carried out by other means.

The most significant point to these modes of simulation is that the only implicit assumptions being made are standard and independent of a design. Thus a designer can be confident that the simulation model validly reflects the design decisions, while providing a variety of tools for behavioral studies. Note that all of these simulation modes may be used on any AIP specification at any point of the development process. This freedom of choice will greatly simplify the designers analysis of the design and prevent many types of complex and expensive errors.

3.4 Real-Time Testing

Even when the simulation model is free of invalidating assumptions, the study of real-time behaviors is complex and expensive. The relative speeds of each process are independent (and continuous-valued) variables that produce too large a space to be blindly surveyed by experiment. We must use information in the specification to aid in the

experiment design and interpretation. The problems in testing an arbitrary specification are truly prohibitive, and we must somehow avoid them.

Some applications may intrinsically require worst-case types of real-time interactions, and we may never be certain of the correctness of such designs. We can isolate such interactions to a few specially treated modules, while doing a much more through analysis of the remainder. Many real-time systems do not require such complexity. If we carefully design them to meet certain constraints, we may greatly simplify our testing problems.

A study of constraints sufficient to simplify the testing problems could start with a classification scheme for real time systems, such that with increasing constraints testing is easier. The designer would then meet these constraints as much as possible in the design. An optimal theory of such constraints has not been attempted as yet. Indeed, it could not even be studied without some formalization of asychronously interacting processes such as our AIP. The following classification is a beginning and will illustrate the approach.

3.4.1 Type of Invariancy

We will first classify real-time systems by the properties that remain invariant as the realization speeds change. These properties are the partial order of the interacting function pairs, and the messages exchanged in each interaction.

Totally Deterministic Real-Time (TDRT)

The behavior of the system is totally determined by the initial state of the system. The only real-time aspects of the system are the performance requirements on the timing of interactions. An example of such a system could be a report generating system, or a pre-programmed real-time controller. "Port-to-port" performance requirements can be factored from the functional specifications into the realization specifications. The behavioral testing of a TDRT specification thus reduces to the simpler case of testing a non-real-time specification by simulation/emulation in any mode and at only one rate. No other behaviors are possible and the real-time aspect of the testing can be done exhaustively.

Pairing Deterministic Real-Time (PDRT)

The partial order of the pairing of the interacting functions is invariant to the realization speed. The actual messages exchanged may change. An example of this type of system could be a real-time clock or a proportional controller whose response is dependent on the contents of the stimulus message. In this case there is an intrinsic functional dependence on relative process rates.

The testing of a PDRT specification could be reduced to the totally deterministic (TDRT) case above for each possible message value. This is essentially the same as non-real-time testing on function argument ranges. PDRT also eliminate the complexity of speed ratios and qualitative changes of behavior with speed changes.

Non-Deterministic Real-Time (NDRT)

Neither the interaction pairing partial order nor the message values are invariant to changes in realization speeds. The behavior of the system may be totally dependent on the realization speeds. An example of a NDRT system could be an adaptive control system that changes modes under some circumstances.

The testing of NDRT specifications may involve the worst case problems. However, the designer might be able to partition the space of the relative rate values into a set of classes such that for each class, the specification becomes either TDRT or PDRT. The testing of the NDRT specification then reduces to the testing of a set of TDRT or PDRT specifications. Any residual partitions that can not be reduced in this manner must be tested the hard way, if at all.

The above partitioning analysis might disclose that some behaviors are unexpected and undesirable. In this case, the designer could introduce additional precedence constraints into the functional specifications that would eliminate that speed dependence and simplify the partition of the relative rates. Proceeding in this manner, the designer could isolate the worst case testing problems to only the truly complex interactions. Perhaps, once identified, many of these complex cases can be eliminated by a redesign for improved testability.

3.4.2 Behavioral Decomposition

We can also simplify the real-time testing problem by factoring the processes of a specified system and studying a part in the context of the possible behaviors of the others. For example, many real-time systems can be divided into environment and controller processes. In the limit, each process in the system could be considered the controller process for the others. The behavior of the system from the point of view of the selected part of the system may be much simpler. A number of special cases lead to significant easing of the real-time testing problems. Some of them will be discussed below.

Totally Non-Interacting Reflex Arcs

The response to a stimulus is a TDRT type behavior with a functional interaction pattern that is independent of the other system activities except for performance degradation in a realization due to resource contention. The performance requirements can be factored into a specification stimulus to response (port-to-port) time and a realization part. The specification requirement part gives isolated performance requirements with a safety factor for realization degradation. The realization requirement part is the limit on permissible degradation due to realization resource contention. The only correlations between the reflex arcs are via the realization resource contentions. Many data acquisition systems are mainly

formed of such arcs, as are multiplexed controllers.

The real time testing of such reflex arcs can be carried out as for TDRT specifications with behaviors restricted to the point of view of the selected part of the system.

Partially Non-Interacting Reflex Arcs

The response to a stimulus is a PDRT type behavior that has an interaction pattern independent of other system activities. Many real-time control systems are almost completely composed of such reflex arcs. The interaction messages themselves may be dependent on the overall state of the system. The performance requirements on the reflex arc may be factored, as above, into a functional stimulus to respond (port-to-port) time and a degradation limit on resource contention in the realization of the system. The only correlations between the reflex arcs are via message values and the realization resource loadings introduced by parallel contention.

The testing of such reflex arcs can be carried as described above for PDRT specifications.

3.5 Conclusions

The extension of the formal specification properties to include performance requirements not only is practical but also can provide new and powerful tools for specification analysis.

We have developed a family of distributed simulator models for simulation of distributed systems that can significantly improve the simulation efficiency for real-time systems while preventing the introduction of many kinds of subtle errors in the model.

We have developed a real time classification scheme that can be used as a basis for real-time testing simplifications and new analysis tools. Real-time testing is so difficult and expensive that a designer must use analyzable interaction structures where possible. The current state of the theory suggests that models for such analyzable structures can be created and used practically.

This work on performance requirements has initiated several important theoretical questions whose resolution could significantly aid in improving the current state of design theory. These questions should be pursued.

4. RSL Analysis (RR-2)

4.1 Property Definitions

We start our analysis by postulating in subsequent sections a series of properties of specifications, each of which must be practically testable on any specification. These properties do not include all properties that could ever be formalized; however, they 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. Furthermore, they have been defined to be generally applicable to any potential discrete system specification. The properties can thus be used to characterize and compare different specification languages.

4.1.1 Formality

In keeping with our emphasis on very large systems we must restrict ourselves to specifications and techniques that can be formally defined -- automated tools and supports being 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.

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. In particular, we must specify approximations to the desired systems formally. Note that 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 a set of wishful prescriptions.

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" which is currently obtained by an ordinary programmer. The utility is obvious.

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

If $R \subseteq R_1 \times R_2$ then we define $R[r_1] = \{r_2 : (r_1,r_2) \in R\}$ and $R[[r_2]] = \{r_1 : (r_1,r_2) \in R\}$.

Let $\mathbb{A}$ and $\mathbb{B}$ be finite alphabets. A specification is a finite string over $\mathbb{A}$. A specification language $\mathbb{L}$ is a set of specifications. A computation is a countable sequence of states, where a state is a finite string over the alphabet $\mathbb{B}$. The computation space $\mathbb{C}$ is the set of all computations. $\mathbb{C}_\infty$ is the set of all infinite computations. Note that we interpret finite length computations to be blocking computations. A system is a set S such that:

(a) S is a non-empty subset of $\mathbb{C}_\infty$

(b) For any state a, $\{a' \in \mathbb{B}^* : \exists x \exists y(\langle x,a,a',y\rangle \in S)\}$ is finite

(c) For any state a, if $\langle x,a,y\rangle \in S$ and $\langle w,a,z\rangle \in S$, then $\langle x,a,z\rangle \in S$.

The system space $\mathbb{S}$ is the set of all systems. An interpreter is a relation in $\mathbb{L} \times \mathbb{C}$. These definitions state that there is a specification language $\mathbb{L}$ whose semantics are sets of computations in $\mathbb{C}$ as defined by $\mathbb{I}$. (Representing the

states of a system as strings over some alphabet entails 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 (c) 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 definition of a system says that any state has only a finite number of successor states. Finally, property (a) says that the system is cyclic, i.e., does not halt. This entails 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 in 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.

In summary, our system definition is very general; it is hard to imagine any "digital system" which could have been left

out. 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 meta-method when performance properties will be added.

4.1.2 Consistency

We will also require that a specification does in fact 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> iff $\Pi[L] \in \mathbb{S}$.

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. Namely, 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 whether system specifications possess this property.

4.1.3 Effectiveness

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

---

[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.

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. 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 corresponds most closely to our idea of "running" a system.

A specification $L \in \mathbb{L}$ is <u>effective</u> iff L is consistent and there exists an algorithm which, given any state σ appearing in a computation in $\Pi[L]$, will produce the set of all states σ' which are immediate successors to σ in computations of S.

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 initial sequences of the 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 can 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.

4.1.4 Modularity and Homogeneity

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 an abstraction of the system specified by the latter. To

---

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

formalize this, we must define "component" and "abstraction."

Let $E \notin \mathbb{B}^*$ and let $\mathbb{C}_E$ be the set of all countable sequences of strings in $\mathbb{B}^* \cup \{E\}$. Let $m_E : \mathbb{C}_E \to \mathbb{C}$ be a mapping that eliminates elements with the value E from computation sequence.

A system S is an <u>embedded abstraction</u> of a system S' iff there exist functions f, p_s, p_c and symbol E such that:

(1) $f : S' \to S$ where f is a bijection

(2) $p_s : \mathbb{B}^* \to \mathbb{B}^* \cup \{E\}$ where $E \notin \mathbb{B}^*$ and p_s is primitive recursive

(3) $p_c : \mathbb{C}_\infty \to \mathbb{C}_E$ where $p_c(\langle a_1, a_2, a_3, \ldots \rangle) = \langle p_s(a_1), p_s(a_2), p_s(a_3), \ldots \rangle$

(4) for any $C' \in S'$ there is a $C \in S$ such that $f(C') = C$ and $C = m_E(p_c(C'))$.

Informally f pairs up computations in S and S'. The function p_s takes the states in the S' computations and either changes some state information or indicates that the state is to be eliminated entirely (by mapping it to E). The function p_c serves merely to apply p_s to each state in an S' computation. The paired computations are related in that $f(C) = C'$ iff $C = m_E(p_c(C'))$.

By way of illustration let S be a system modeling execution of a program (where 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 (where its states are machine states, and its steps are instruction executions). Then S is an embedded abstraction of S', with p_c 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.

The definition of a system S being an abstraction of a sytem S' reads the same as the definition of embedded abstraction except that f need only be 1-1.

A component K is a finite string over the global alphabet A. The component language $\mathbb{K}$ is a set of components. Within a program a component might be any substring generated from a nonterminal of the context-free grammar.

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 and is also a component.

An abstraction relation $R_A \subseteq \mathbb{K} \times \mathbb{K}$ contains a pair (K_1,K_2) only if for any $L_2 \in R_{\mathbb{K}}[[K_2]]$, the string L_1 (formed by textually replacing any one occurrence of K_2 in L_2 by K_1) is in $\mathbb{L}$ and either (a) one of L_1 or L_2 is not consistent or (b) L_1 and L_2 are consistent and $\mathbb{H}[L_1]$ is an abstraction of $\mathbb{H}[L_2]$.

The reason that L_1 can be inconsistent, even though K_1 is a valid abstraction of K_2, is that consistency is intrinsically global. For instance, K_1 might be a primitive

function, and K_2 might be an elaborated version in which an interaction with another part of the system (perhaps to obtain control information) is specified. Then substituting K_1 into a consistent specification containing K_2 (which must itself have a specification on the other half of the interaction in order to be consistent) will create an inconsistent specification, in which the other half of the interaction is left hanging.

A specification $L \in \mathbb{L}$ is <u>modular</u> iff L is consistent and $R_{\mathbb{K}}[L]$ is not empty, and for any $K \in R_{\mathbb{K}}[L]$, the set given by $R_A[[K]] - \{K\}$ is not empty. A specification $L \in \mathbb{L}$ is <u>homogeneous</u> iff L is modular and for any $K \in R_{\mathbb{K}}[L]$ and $K' \in R_A[K]$, the specification L' (formed by textually substituting K' for any one occurence of K in L) is consistent. Informally, homogeneity says that for $L \in \mathbb{L}$ any substitution defined by R_A leaves us with a consistent specification.

4.1.5 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 meta-method.

An informal attribute set T is a finite string over the global alphabet $\mathbb{A}$. An 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 specification $L \in \mathbb{L}$ is informally extensible iff for every $K \in R_{\mathbb{K}}[L]$ and every $T \in R_{\mathbb{T}}[K]$, the specification L', formed by substituting T' for T in L, is such that $\Pi[L] = \Pi[L']$.

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 will not provide much assistance other than that of a controlled data base, but on the other hand we will not hinder that which can be done by the designer. In conclusion, 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.

4.1.6 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 are unique 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.

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 of 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.

The asynchronous combination of systems $S_1,\ldots,S_n$ is the system whose computations are sequences of states $\langle A_1,A_2,A_3,\ldots\rangle$ where:

(1) A_1 encodes $\langle {}_1a_1,\ldots,{}_na_1\rangle$ and for any $i \in \{1,\ldots,n\}$ we have that ${}_ia_1$ is an initial state (first element) of some computation sequence in S_i,

(2) for any $i > 1$, A_{i+1} encodes $\langle {}_1a_{i+1},\ldots,{}_na_{i+1}\rangle$ and A_i encodes $\langle {}_1a_i,\ldots,{}_na_i\rangle$ where for some $j \in \{1,\ldots,n\}$ we have

(a) ${}_ka_i = {}_ka_{i+1}$ for $k \in \{1,\ldots,n\}$, $k \neq j$,

(b) ${}_ja_{i+1}$ is a successor state of ${}_ja_i$ in system S_j.

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.

Asynchronous Specifications

A specification $L \in \mathbb{L}$ is asynchronous iff L is consistent and there are $n \geq 2$ consistent specifications $L_1,\ldots,L_n \in R_{\mathbb{K}}[L]$ (where $L_1,\ldots,L_n$ are disjoint substrings

of L) such that the asynchronous combination of $\Pi[L_1],\ldots,\Pi[L_n]$ is an abstraction of $\Pi[L]$.

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 sytems ([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 multi-programming is fixed or bounded.

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

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.

4.1.7 Generality

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

A specification language $\mathbb{L}$ is <u>complete</u> if for every $S \in \mathbb{S}$, there exists an $L \in \mathbb{L}$ such that $\Pi[L] = S$. This is not a product property at all, but rather a property of a design process. It seems that completeness needs to be proved in a theorem 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.

4.1.8 Conclusions

We have defined an abstract set of properties for specifications and specification languages. A suitable specification language for distributed systems must have at least these critical properties in a useful form. In fact, we may use these properties as requirements for specification language design. Further, we will clearly add further properties as required to support subsequent development of our design theory. These will be introduced subsequently.

Figure 4.1 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."

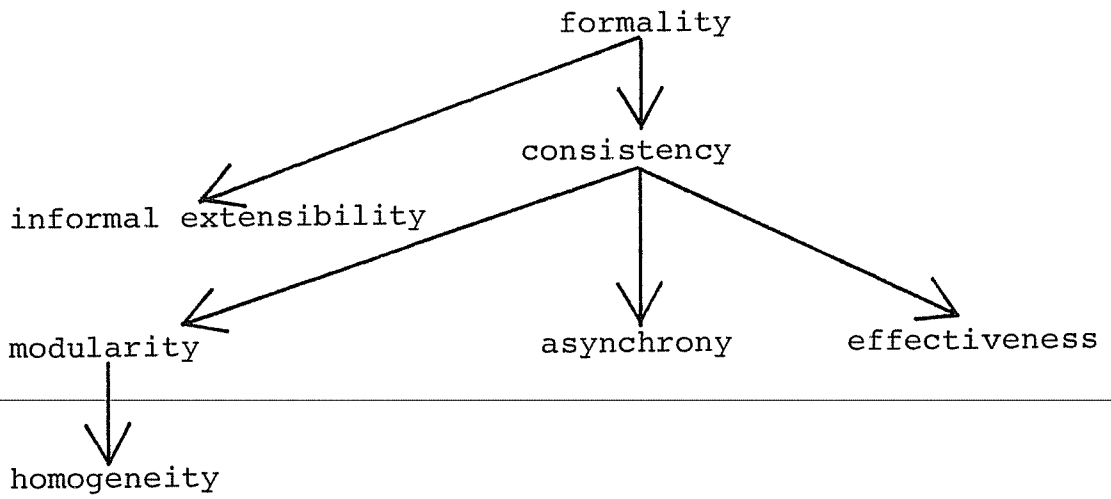


Figure 4-1: The required product properties.

The property definitions 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, automability, 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. On the other hand, 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.

4.2 RSL Property Tests

We set out here to determine for any $L \in \mathbb{L}$ whether or not we can test for the following properties: informal extensibility, consistency, effectiveness, modularity, homogeneity, and asynchronicity.

4.2.1 Definitions

Before we conduct the analysis we must define the following:

(1) specification language $\mathbb{L}$,

(2) state of a computation,

(3) state successor function,

(4) interpreter $\mathbb{I}$,

(5) component language $\mathbb{K}$,

(6) informal attribute set $\mathbb{T}$,

(7) abstraction relation R_A.

Once we have done 1-7 we can then perform the RSL analysis.

Define Specification Language

Actually we will define two languages. We define $\mathbb{L}_1$ to be the set of RSL sentences as defined by Appendix D of the REVS User Manual [MD]. We define $\mathbb{L}_2$ to be the set of RSL sentences with their associated drivers and exogenous event routine. Thus $\mathbb{L}_1$ is a pure RSL language while $\mathbb{L}_2$ sentences have both RSL and Pascal. This would seem to exhaust the reasonable choices of "RSL specification language." We will see

that each language has its drawbacks.

Define State of a Computation

The state in a computation generated by an $\mathbb{L}_1$ specification contains an encoding of all data, files, entities, time, and the event calendar. The state in a computation generated by an $\mathbb{L}_2$ specification contains an encoding of not only all data, files, entities, time, and the event calendar, but also the states of the drivers, the exogenous event routine, and any other user-defined or system procedures. The initial state is as defined by RSL and Pascal.

Define State Successor Function

For both $\mathbb{L}_1$ and $\mathbb{L}_2$ specification languages we give a choice of two state successor functions. In choice (1), we pass from one state to the next state via the evaluation of an R-net $(\mathbb{L}_1, \mathbb{L}_2)$, a driver $(\mathbb{L}_2)$, or the exogenous event routine $(\mathbb{L}_2)$ as defined by REVS and Pascal. The state successor function is that mapping that applies an R-net, driver, or exogenous event routine to a state. In choice (2), we assume that time is discrete. We pass from one state to the next state when all computation at one time is completed. The state successor function is that mapping that applies the R-nets, drivers, and exogenous event routine until nothing more is left to do at one time so we can advance to the next time. These two choices would seem to exhaust the reasonable choices of successor function.

Define Interpreter

Because we have two languages and for each language we have two choices for state successor function, we will define four interpreters. Let $\mathbb{I}_{2,1} \subseteq \mathbb{L}_2 \times \mathbb{C}$ be the interpreter for specifications in $\mathbb{L}_2$ using choice (1) for state successor function. Let $\mathbb{I}_{2,2} \subseteq \mathbb{L}_2 \times \mathbb{C}$ be the interpreter for specifications in $\mathbb{L}_2$ using choice (2) for state successor function.

We next define $\mathbb{I}_{1,1} \subseteq \mathbb{L}_1 \times \mathbb{C}$ (the interpreter for specifications in $\mathbb{L}_1$ using choice (1) for successor function). First we assume that the alphas are functional, that is, they are some primitive mapping from inputs to outputs. Now, note that the specification has not specified enough information on the order and timing of R-net execution; the drivers and exogenous event routine are not specified. So, to get one of the computations in $\mathbb{I}_{1,1}$, we choose some recursive set of drivers, a recursive exogenous event routine, and recursive functions to replace the primitive alphas and then run that $\mathbb{L}_2$-like specification on a simulator to get a computation. We do this for all legal choices of recursive drivers, exogenous event routine, and functions to get all the computations generated by the $\mathbb{L}_1$ specification. We can similarily define $\mathbb{I}_{1,2} \subseteq \mathbb{L}_1 \times \mathbb{C}$: the interpreter for specifications in $\mathbb{L}_1$ using choice (2) for state successor function.

Define Component Language

We will define one component language $\mathbb{K}$ for both $\mathbb{L}_1$ and $\mathbb{L}_2$. This means that the drivers and exogenous event routine will not be part of any component. This is no serious restriction as it is very difficult to define any general form of substitution of one Pascal driver for another Pascal driver.

The actual components will be strings of terminals (occurring in some specification in $\mathbb{L}_1$ or $\mathbb{L}_2$) that are derived from the nonterminals ⟨node⟩, ⟨new element definition⟩, and ⟨element definition sentence⟩ appearing in Appendix D of the REVS Users Manual. Basically, ⟨node⟩ derives the different kinds of nodes in an R-net; ⟨element definition sentence⟩ derives the attributes, relations, paths, and/or structures associated with a given element; ⟨new element definition⟩ derives the definition of an element. It seems reasonable that the component language $\mathbb{K}$ include at least these strings.

Define Informal Attribute Set

Again, we will define one informal attribute set $\mathbb{T}$ for both $\mathbb{L}_1$ and $\mathbb{L}_2$ (causing us to ignore the informal attributes associated with Pascal). Let $\mathbb{T}$ be the set of all strings of non-quote characters enclosed within quotes that do not represent a Pascal routine. It is important to note that the elements of $\mathbb{T}$ embedded within components do not affect the interpreter.

Define Abstraction Relation

Before we describe the actual abstraction relation, we begin with a comment about RSL. RSL was designed for specifcation at individual points in the requirements development process. They do not discuss transformations of specifications that allow us to go from one point in the development process to another while guaranteeing subset or superset behavior. This is roughly what the abstraction relation addresses. The design of the abstraction relation ideally should have been done by the original RSL language designers. Such transformations can be a difficult task when the alphas are procedurally interpreted.

For the purposes of this analysis we will describe a very simple abstraction relation. Let $(R_1,R_2) \epsilon R_A$ where R_2 is a legal R-net definition and R_1 is a legal R-net definition that refers to no more global items than R_2 can. Further, R_1 differs from R_2 only in that before some node that is not the first node in an R-net we insert an or-node and some non-blocking extra paths from the or-node.

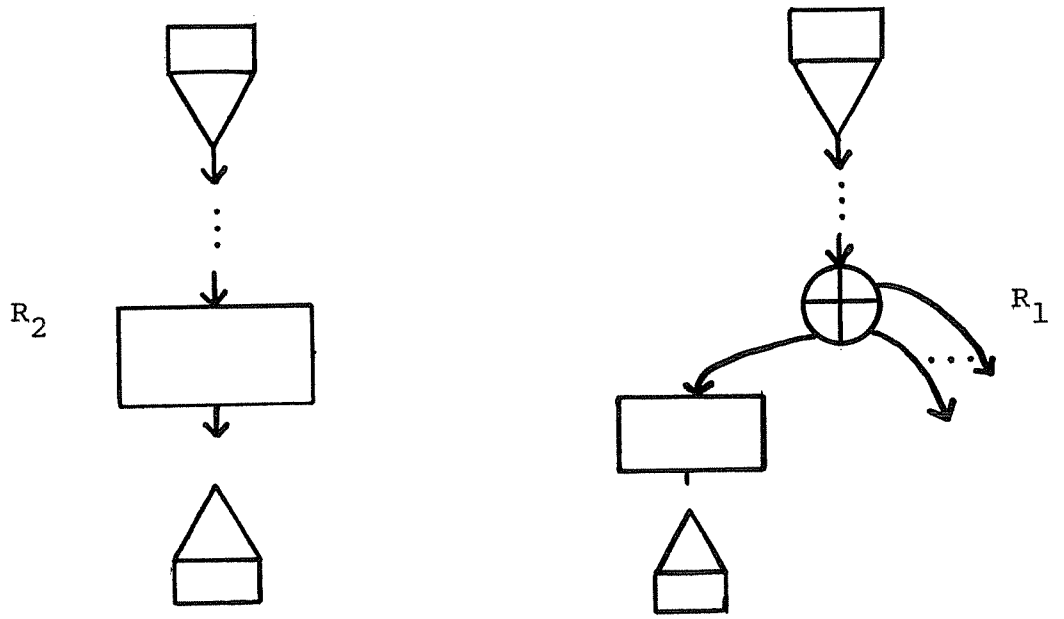


Figure 2-1. Insertion of or-node and additional non-blocking paths into a single path.

We can add other pairs to R_A without altering later proofs provided:

(**) for any $K \in \mathbb{K}$, we can test if $R_A[[K]]$ is empty or not, and

(***) if $K \in R_{\mathbb{K}}[L]$, $(K',K) \in R_A$, and $\Pi[L]$ is a system, then $\Pi[L']$ is a system, where L' is formed by substituting K' for K in L.

4.2.2 Analysis

We remind the reader that the properties we wish to apply to RSL are: formal, informally extensible, consistent, effective, modular, homogeneous, and asynchronous. The main

concern of our analysis is to determine whether or not we can test any specification for these properties. We will only attempt crude estimates of how many specifications actually have a particular property (e.g., none, some, all).

Formal

We assume that all specifications in both $\mathbb{L}_1$ and $\mathbb{L}_2$ are formal because we have given a definition of a specification language, interpreter, component language, informal attribute set, and abstraction relation.

Informally Extensible

Every specification in either $\mathbb{L}_1$ or $\mathbb{L}_2$ under any of the defined interpretations is informally extensible because the informal attributes do not affect the simulation. So then we can test for informal extensibility.

Consistent

For $\mathbb{\Pi}_{1,1}$ we can test for consistency. Unfortunately, this is because no specification is consistent. For any $L \in \mathbb{L}_1$ there are finite length (blocking) computations for the SSSTARTUP driver that schedules nothing on the calendar. Therefore $\mathbb{\Pi}[L]$ cannot be a system and L is not consistent. Note that the other properties of effective, asynchronous, modular, and homogeneous all depend upon a specification being consistent. Therefore under $\mathbb{\Pi}_{1,1}$ we can test for these

other properties because no specification in $\mathbb{L}_1$ has them.

For $\mathbb{\Pi}_{1,2}$ we can test for consistency, unfortunately because no specification is consistent. For any $L \in \mathbb{L}_1$ there are finite length (blocking) computations for the SSSTARTUP driver that schedules the exogenous event routine which schedules itself with no delay, which possibly schedules other calendar events with no delay, and so on ad infinitum. Time never advances so a next state is never reached. So $\mathbb{\Pi}[L]$ is not a system and L is not consistent. See the note in the previous paragraph.

For $\mathbb{\Pi}_{2,1}$ and $\mathbb{\Pi}_{2,2}$ we cannot test for consistency. The Pacal drivers allow us to reduce the halting problem to testing for consistency.

Effective

The analysis will be done for $\mathbb{L}_2$ only. Now we claim that $L \in \mathbb{L}_2$ is effective iff L is consistent. If L is effective then by definition it is consistent. If L is consistent, there are a finite number of next states. The definition of RSL and Pascal gives us an algorithm to generate the next states (and allows a simulator to be construced for RSL and Pascal). So then a test for effectiveness is equivalent to a test for consistency in $\mathbb{L}_2$.

Modular

The analysis will be done for $\mathbb{L}_2$ only. If we can test for consistency then we can test for modularity. The testing procedure would parse an RSL sentence L and find the components. If $R_{\mathbb{K}}[L]$ is empty then L is not modular. Otherwise for any $K \in R_{\mathbb{K}}[L]$, we test whether or not $R_A[[K]]$ is empty. (We can do this by (**).) If there is an empty $R_A[[K]]$ then L is not modular. Otherwise L is modular.

Homogeneous

The analysis will be done for $\mathbb{L}_2$ only. Now we claim that $L \in \mathbb{L}_2$ is homogeneous iff L is modular. If L is homogeneous then L is modular by definition. If L is modular then by (***) we are guaranteed that L is homogeneous. So then a test for homogeneity is equivalent to a test for modularity.

Asynchronous

The analysis will be done for $\mathbb{L}_2$ only. Yes we can test for asynchrony, unfortunately because for a trivial reason no specification is asynchronous. More than two specifications can never be disjoint substrings of one specification because in each specification there is exactly one copy of the SSSTARTUP driver.

Even if we assume all the syntactic difficulties away, due to the nature of RSL the sub-specifications must be

totally isolated from one another. The sub-specifications would have no interactions; this would not be an extremely useful notion. A more serious difficulty is that when trying to partition R-nets up into separate specifications, it is undecidable if they both use (e.g., read or write) a global variable. We could then devise a specification that has two sub-specifications iff two R-nets do not use a global variable.

4.3 Conclusions

There are some deficiencies that stand out in RSL. First is the use of Pascal to specify the external environment. Pascal is too unconstrained to guarantee many properties. Secondly, RSL has no explicit encoding of interactions. RSL does not have asynchronously interacting specifications. Thirdly, RSL does not discuss transformations on specifications that guarantee subset or superset behavior. This potentially could be of great help to RSL users. The results are summarized in Table 4-1.

Property	Interpreter	Testable?	Testable assuming consistency?	How many specs have this property?
IE	Any	Yes	Yes	Some
C,E,M,H	$\Pi_{1,1}$ $\Pi_{1,2}$	Yes	Yes	None
C,E,M,H	$\Pi_{2,1}$ $\Pi_{2,2}$	No	Yes	Some
A	Any	Yes	Yes	None

Table 4.1 Summary of analysis of RSL/REVS for formal properties.

The above deficiencies in RSL from the point of view of distributed systems are not surprising and do not reflect deficiencies in RSL designers. Rather, they reflect the fact that RSL was intended for non-distributed system applications and tailored to these applications. It is precisely in the area of asynchronous "internal" interactions that distributed control systems differ markedly from centralized applications.

5. Task RR-3: MRSL Design

5.0.1 MRSL Semantics

One important task which we have undertaken as part of our current analysis of RSL is to determine how the language might be modified in order to insure that the new language would possess the formal properties described in sections 3 and 4. We have used our formally defined asynchronously interacting processes (see the paragraphs below for definitions and explanation) as the entities to be modelled by the modified RSL (referred to concisely as MRSL). Several methods for modifying RSL have been examined, either by restricting RSL syntax or by augmenting it. In either approach the resulting MRSL could be used as a source language for writing specifications, or alternatively, the specifications in MRSL could be generated from specifications in another source language. In any case the semantics of MRSL will differ from that of RSL since constructs for synchronization of message passing between R-Nets are lacking in RSL and yet are essential for modelling AIP. A brief, informal description of the semantics of MRSL immediately follows our formal definitions of AIP in this same section.

We first define individual processes. A process state is a string over $V \subseteq A$. A process state space is the set of all process states. A state successor relation is a mapping from a process state space to a process state space. An isolated process is a tuple (f, s) of successor function and state space. A process

specifies a system by defining a generator of its computations (f, s) specifies

$$S = \{c:\forall s_i \in s,\ c = \langle s_i, f(s_i), f^2(s_i), \ldots \rangle\}.$$

Each function may have attributes of evaluation time interval and space. Each non-primitive function and set will be defined as expressions of primitive functions and sets. Finally, each primitive function and set may be defined as expressions of still more primitive functions and sets.

We now define asynchronously interacting processes (AIP). Let each process (f_i, s_i) specify an interacting system S_i. The set of asynchronously interacting processes $((f_1, s_1), \ldots, (f_n, s_n))$ specifies the interacting combination of the S_i. The only effect of interactions is to eliminate some of the computations in the asynchronous combination of the systems S_i. Thus we can study isolated AIP confident that they will generate no new behavior when combined with others.

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

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

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

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

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

An important conculusion of our previous contract work was that asynchronously interacting processes (AIP) are a powerful way to define DDP requirements, designs and implementations. AIP also form a suitable basis for the development of an extensive and highly automated design methodology.

RSL allows the specification of "interactions" via interfaces to the "outside" (non-RSL described) world. These interfaces define only the message types passing the interface and do not model synchronizations or message corrections as required for DDP specifications. Further, no "internal interfaces" exist beyond those implied by uncorrelated use of shared variables.

If we are to model AIP, using RSL, we must re-interpret the semantics of an RSL interface and add some new attributes to messages. We can then model an exchange function by a coupled pair of interfaces, one to represent the initiation of an interaction (termination of an R-Net) and another to represent the completion (initiation of an R-Net) of an interaction.

The drivers associated with interfaces now simply model the exchange interactions and become standard modules, independent of the specific application. We thus introduce a standardized interaction scheduler that will initiate events for the usual task scheduler.

Each AIP will now be represented by a set of R-Nets operating on shared data local to that process and interacting via the re-interpreted interfaces with other R-Nets in the same or different processes. Each R-Net defines the non-interacting processing from R-Net initiation event (all interactions are represented as scheduled events) to R-Net termination event.

The state successor function that is defined in terms of exchange functions will be decomposed into a set of functions that may interact only at their initiation and termination. These functions can be grouped into R-Nets such that the R-Nets have interface initiation and termination.

The problem of how to model asynchronously interacting processes (AIP) via RSL/REVS [MD] has been studied in some detail. The principal concerns lie in (1) choos-

ing a method for representing AIP in RSL/REVS, (2) providing automated analysis tools to insure that the formal properties described in Section 3 are satisfied by the specifications, and (3) generating all necessary code for the REVS simulation of specifications satisfying these properties. In the remainder of section 5.0 we are concerned mostly with examining the merits of various possible ways of representing AIP in RSL/REVS. For any choice of representation it is a relatively straightforward matter to provide the additional standardized software necessary to perform a simulation for any given set of specifications.

We now discuss four possible schemes for modelling AIP with RSL/REVS software (in sections 5.0.2 to 5.0.5). The first three of these (table model, message model, and augmented RSL model) employ modifications to RSL/REVS to produce source languages for specifying AIP, and so these languages are referred to collectively as modified RSL, or MRSL. (A source language based on a mixture of the three techniques is also possible but not discussed here.) The last scheme, the AIP model, employs a subset of RSL/REVS merely as an intermediate language, so that the original source language for defining AIP would need to be translated into RSL/REVS. The AIP model (section 5.0.5) has already received some testing by way of the prototype examples in section 5.2. Results have provided significant insight into the feasibility of RSL/REVS as a vehicle for representing AIP, and these observations are reported in sections 5.0.6 and 5.4.

5.0.2 Table Model

It has been noted in the beginning of section 5 that RSL/REVS does not have any explicit provision for representing message passing between AIP, whose computations are modelled by R-Nets (see [MD]), and likewise synchronizations associated with message passing. In fact, the concept that a collection of R-Nets models an individual AIP is not expressible in RSL/REVS. A principal motivation for MRSL has been to remedy this situation. One intuitively simple solution is to store the information not semantically expressible by RSL syntax in (non-RSL) variables accessible to the simulation procedures of REVS. Data structures consisting of list and pointers (hence the name "table model") can then be used to keep track of pending messages, message contents, activations and scheduling of R-Nets, simulation timings, and so forth. Both static and dynamic information, both types of which require initialization, are involved.

With the table model the drivers (see [MD]) for individual R-Nets can be of a uniform structure, where the only differences in code (Pascal, of course) are due to the size of respective messages transferred between R-Nets. A standard exogenous event routine (see [MD]) then makes access to the simulation information in the data structures between R-Net activations in order to decide which messages to exchange and which R-Nets to activate next. (The code of the exogenous event routine changes depending on the

number of R-Nets in the simulation, but this amounts only to a kind of parameterization.) It is important to note that all the code for both drivers and exogenous event routine can be automatically generated from the combined RSL (i.e., strictly R-Net) specifications and the user-supplied data structures accessible to the simulation routine.

The main drawback of the table model is that the designer must not only perform by hand the highly specialized AIP decomposition described in section 5.1.2, but must also be fully cognizant of the actions of the simulation routines, for example, the synchronization and activation aspects of simulation. (The complex detail makes even the simplest examples too unwieldy for presentation here.) Thus a great deal of the designer's effort must be expended in decomposing AIP for R-Net modelling and then restoring the interrelationships between individual R-Nets by filling out data structures with coded information. The designer can, of course, be aided by analysis tools which discover errors in the R-Nets and data structures so obtained. However, such aid does not eliminate the inconvenience and difficulty of initially obtaining specifications which can be analyzed. On the other hand, this is an improvement over conventional design in RSL/REVS, since only a subset of RSL is used and details of REVS simulation routines are shielded from the designer through a specialized encoding (in the simulation data structures) from which simulation

routines can be generated automatically. In either case the designer must be conversant in Pascal so that detailed AIP computations can eventually be simulated (specifically by the alpha nodes in R-Nets). In the AIP model (section 5.0.5 below) a homogeneous specification language to replace the combination of RSL (with or without supplementary data structures) and Pascal as source languages will be discussed.

5.0.3 Message Model

Our second technique for modelling AIP is named "message model" by analogy with the table model above, namely in that information contained in the (non-RSL) data structures in the table model is instead included within RSL messages (in addition to the actual AIP message contents) in the message model. In this way both the static and dynamic information discussed in section 5.0.2 is passed back and forth between the R-Nets and simulation routines. This technique eliminates the data structures of the table model and permits AIP specifications to be written entirely in RSL (and Pascal for alphas, of course). As before, all necessary parameterization of standardized REVS simulation routines can be performed automatically. Thus the message model merely shifts the mode of handling information necessary for analysis and simulation, and so the analysis tools developed for the table model would not differ significantly from those in the message model. However, the inconvenience to the designer in writing specifications is even greater

than in the table model because he must now encode data structures somewhat clumsily and redundantly in the body of messages. Efficiency in simulating the specified AIP also decreases because variables must be passed from the RSL routines where they are encoded and then must be copied over by the simulation routines, possibly altered, and finally returned to the RSL routines whenever an AIP message (i.e., true message) is sent.

5.0.4 Augmented RSL Model

By augmenting the syntax of RSL to create a new type of MRSL we can express all the necessary relationships between R-Nets and their messages in MRSL syntax alone. This is in contrast to coding the same information in (non-RSL) data structures (table model) or R-Net messages (message model). However, as in the previous models the drivers and exogenous event routine could be generated automatically for the purposes of simulation. Furthermore, the additional MRSL syntax features could be translated into a combination of data global to all simulation routines and data local to drivers. (This arrangement is somewhat simpler than the coding of simulation data in the table model and does not require designer intervention.)

The RADX data base and anlaysis package of REVS could be used to extract the syntactically expressed R-Net and message relationships from the MRSL source language. However, RADX is completely inadequate for the purposes of

analyzing specifications for compliance with the formal properties which they must exhibit (sections 3,4). Thus RADX would be superfluous in our analysis scheme since the entire MRSL source code would have to be considered, not only with respect to the new language constructs of MRSL but also those shared with conventional RSL. Very importantly, though, REVS software does permit new language features to be added to RSL without affecting the ability of the remainder of the REVS package to perform simulations, as with conventional RSL. For the type of RSL modification anticipated here, the new MRSL syntax features would essentially be ignored by REVS for the purposes of simulation. The work of interpreting MRSL in the augmented RSL model would then depend upon non-REVS software to a slightly greater degree than in the table model or message model. That is to say, in the table model and message model non-RSL code is required primarily for semantic checking of MRSL specifications and generation of parameterized simulation routines. In addition, in the augmented RSL model new syntax constructs must be translated into data for the simulation routines.

Extensions to RSL syntax would presumably be in the form of new elements, relationships, and attibutes (see [MD]) with self-explanatory or otherwise suggestive names for the aid of the designer. However, in spite of the homogeneity of the specification medium the designer still must be able to perform the same AIP decomposition as in

the table model and message model and also must be familiar with the same technical details of R-Net activation and synchronization as in those two models.

5.0.5 AIP Model

If we choose to use RSL/REVS only as an intermediate language in writing AIP specifications then we are free to use a source language totally independent of R-Nets and simulation routines. Instead the AIP specifications would be translated into RSL/REVS without designer intervention so that the specifications could again be translated into simulatable form by REVS. Checks for compliance with the formal properties discussed in section 3 could be made directly on the source language in the AIP model. We could expect the analysis tools in this case to be simpler than in the models of sections 5.0.2-5.0.4 since the source language would be deliberately chosen to reflect more closely the intended AIP. Furthermore all of the following could be performed automatically by reference to the source language: AIP decomposition, generation of R-Nets themselves (unlike previous models), and generation of all necessary simulation routines. Thus in the AIP model the designer need not know anything about RSL/REVS software, the technical details of message passing, or AIP decomposition. He is then free to write AIP specifications in a language designed for that purpose and unbiased with respect to the software employed for subsequent analysis

and simulation. In addition, practical experience in writing specifications in the AIP model as opposed to the table model (section 5.0.2) shows tha AIP model to be much more concise, in fact, by a factor of at least ten in the sheer amount of source code which must be written for the same AIP.

5.0.6 Comparison of the MRSL Models

Comparison of the four models above for specifying AIP (sections 5.0.2-5.0.5) shows that from the point of view of ease and comprehensibility for the designer that the AIP model is by far superior. This results from the simplicity of syntax and semantics possible in an AIP-oriented source language, which can be far more concise than any MRSL language. Likewise, the analysis of specifications for the required formal properties, such as consistency and completeness, is simpler in the AIP model. Finally, it is to be translated into a subset of RSL/REVS and so does not necessitate any change in the RSL/REVS software. However, it is the most complex of the four models from the standpoint of implementation.

Of the remaining three models the augmented RSL model uses the most homogeneous source language and hence is more comprehensible to the designer than the table model or message model. All three models are roughly of the same difficulty to implement, although the table model is slightly simpler in this respect. (As we have already mentioned, the AIP model has been used for testing of

the prototype examples of section 5.1 and Appendix A.) Each of these three models (and also the AIP model) allows standardized simulation routines to be essentially parameterized for any particular set of MRSL specifications. Thus the differences among the three models lie primarily in the method by which the respective models store and handle information on the scheduling and synchronization of R-Nets based upon the messages they pass to each other. The remainder of section 5 will detail and formalize many of the specification and simulation concepts found in this overview. Further conclusions are drawn in section 5.4.

5.1 Generation of MRSL Structures

Several alternate methods for modifying and reinterpreting RSL/REVS to serve as a medium for specifying asynchronously interacting processes (AIP) were described informally in sections 5.0.2-5.0.5. The name MRSL (modified RSL) was applied to each resulting specification model. In the AIP model MRSL would serve only as an intermediate specification language while some source language such as that developed in section 3.4.3.2 of [Fi322] would serve as the primary source language for

---

[Fi322] Fitzwater, D. R. "The Formal Design and Analysis of Distributed Data-Processing Systems," CSTR 322, University of Wisconsin-Madison, April 1978.

specifying systems. In this model then the MRSL specifications could be further translated by REVS software for the purposes of testing and simulation. Since the semantics of the source language of [Fi322] has been already (informally) stated and example of its use have been presented, it is convenient to indicate the semantics of MRSL simply by translating AIP specifications into MRSL. This is especially true because RSL/REVS has no syntactic or semantic contructs corresponding to process definitions and hence interprocess interactions and synchronizations (at least under the formal definitions employed here). The result is that the semantics of MRSL (in any model) is somewhat obscured by the artificial and rather arbitrary structures and conventions that must be introduced in order to represent AIP. Finally, MRSL in the AIP model provides an excellent framework for explaining MRSL in the other models because the differences are primarily notational in character.

The remaining subsections of 5.1 deal with translations of AIP specifications into that portion of MRSL which can be represented strictly in RSL. This includes the structures necessary for the evaluation of the state successor functions for individual processes exclusive of the evaluation of exchange functions. Evaluation of exchange functions, synchronization of computations specified in MRSL, and all other details relevant to the simulation of the original AIP specifications

will be presented in section 5.3 dealing with MRSL implementation.

5.1.1 AIP Precedence Relations

We will focus now on transformations of AIP specifications into MRSL specifications which can be directly modelled in REVS. More specifically we will detail a sequence of transformations which need to be applied in sequence in order to go from AIP to MRSL specifications. It is important to note that there are many possible ways of writing MRSL specifications and furthermore within each of these there are in addition many satisfactory ways to perform the necessary transformations on AIP specifications. The transformations chosen here are thus quite arbitrary but have been chosen with the particular goals of simplicity, clarity, and ease of implementation. Similarly a great deal of use has been made of graphical methods in order to draw attention to the features of greatest interest and to convey the effect of applying the transformations more clearly than in words alone. Note will be taken wherever specification information not critical to the transformation has been omitted. It should be understood, of course, that all such non-critical information will remain unaltered by the transformations in question. The initial discussion draws heavily on material from section 3.4.3.2 of [Fi322].

In an AIP specification the processes are defined by a state successor function which may be arbitrarily complex and may involve one or more definition statements in the AIP

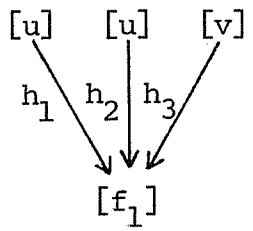
language. If the state successor function consists of more than a single primitive function (which is not elaborated any further than the specification of its domain and range) than it can ultimately be described in terms of primitive functions (including exchange functions) which must be evaluated in some order consistent with the definitions of the process. For example for the state successor function h, where $h(x,y) = f(g(x),g(y))$, the evaluation of $g(x)$ and the evaluation of $g(y)$ may be done in any relative order, including in parallel, but each must be evaluated before $f(g(x),g(y))$ can be evaluated. The immediate goal of this section is, in fact, to show how such precedence information can be inferred individually for each process by means of its definitions.

Since we will be mainly concerned with flow of control during simulation of AIP specifications we will represent function definitions (which are little more than ordinary algebraic expressions) by means of directed graphs. For example, the definition:

Let $f_1 \rightarrow N \times N \rightarrow N \times N \times N$ where

$$f_1(u,v) \equiv (h_1(u),h_2(u),h_3(v))$$

can be rendered graphically as simply:

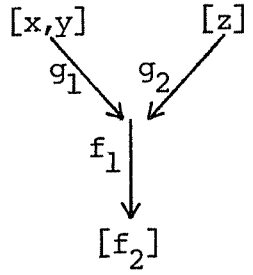


Similarly, the definition:

Let $f_2 : N \times N \times N \rightarrow N \times N \times N$ where

$$f_2(x,y,z) \equiv f_1(g_1(x,y),g_2(z))$$

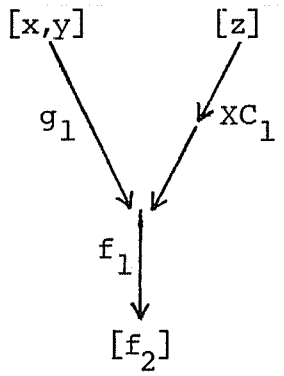
can be rendered as:



Here the symbols enclosed in square brackets are not the names of vertices but instead merely labels associated with them. (There is no need to name vertices in this somewhat informal presentation.) Similarly, arcs need not have distinct function names associated with them. Intuitively, the direction of the arcs shows that in the first example h_1, h_2, and h_3 must be evaluated in order for $f(u,v)$ to be obtained. Similarly, in the second example g_1 and g_2 must be evaluated before f_1 which yields $f_2(x,y,z)$. Arcs will in later discussion be labeled with a function name if they denote computation or, in cases to be explained later, they will be unlabled and will

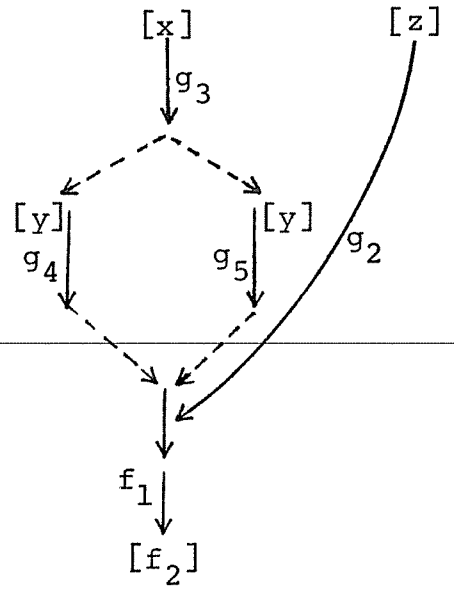
merely express precedence information. These latter arcs will be called "dummy" arcs for obvious reasons. Vertices will generally remain unlabeled except where exchange functions are involved, and the role of these special vertices will be described shortly. It should be noted that a great deal of information is not indicated in the graphs, for example, the domains and ranges of functions. Furthermore, even though the function arguments have been shown in their correct order in the figures above, this information is not encoded as part of the graph. Thus the graph does not have any information on the order of function arguments. However, all information lost is irrelevant to the subsequent discussion of 5.1 (which depends primarily on flow-of-control information) and has been intentionally ignored in order to highlight features of interest. On the other hand all such information is to be subsequently encoded in the final MRSL specifications.

Exchange functions will be represented differently from other types of primitive functions (e.g., primitive functions in the examples above), in that two arcs are employed, where the vertex joining them (rather than an arc) is labeled with the exchange function name. For example if g_2 were instead XC_1 in the second example above then we would have the following graph for f_2:

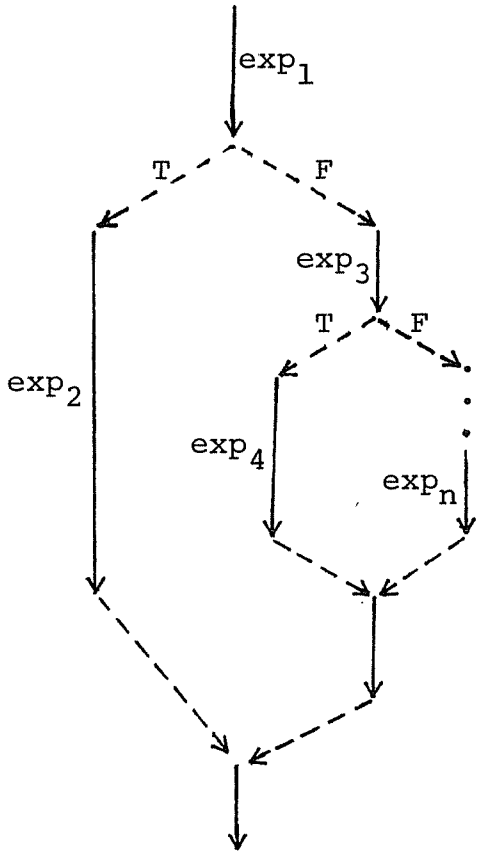


Note that the arcs incident on the vertex labeled XC_1 are dummy arcs. These are included purely for convenience in later graph transformations, and eventually many of these arcs will be discarded.

Selector functions will be drawn as follows where we substitute for g_1 above the selector function $[g_3(x) : g_4(y), g_5(y)]$, that is, "if $g_3(x)$ then $g_4(y)$ else $g_5(y)$":

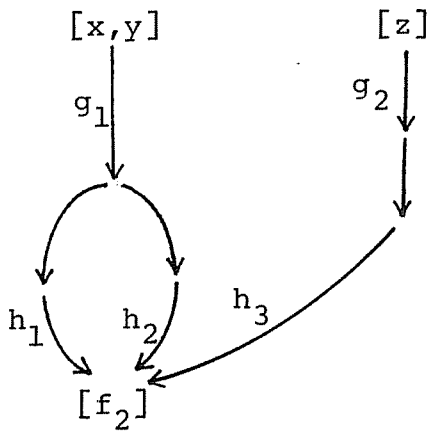


If the selector function had had more paired expressions (e.g., [exp_1 : exp_2, exp_3 : exp_4,...,exp_n]) then they would take the following graphical form:



where the dotted line indicate that additional pairs of expressions may be substituted. The dashed arcs must always occur in multiples of four by construction, and they will be considered as distinct from other arcs for the purposes of transformation. They indicate simply that only one path following the bifurcating dashed arcs will be evaluated, depending upon the Boolean result of the expression preceding these arcs. The labels for these arcs (e.g., exp_1 and exp_3 above) will be omitted whenever the expression is simply a constant.

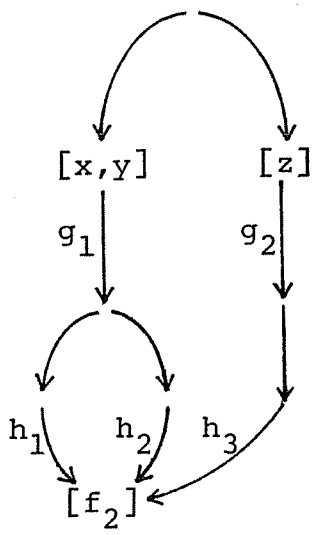
It is now possible to combine precedence information from disparate graphs in order to express state successor functions entirely in terms of primitive functions. This is done by successively eliminating, via substitution, all non-primitive functions from our graphs. For example, if f_2 above is considered as a state successor function and g_1, g_2, h_1, h_2, and h_3 are all primitive then we need only eliminate the non-primitive f_1. This is done by drawing dummy arcs from each argument of f_1 in the definition of f_2 to the corresponding parameters in the definition of f_1 to yield:



Note that $g_1(x,y)$ is used in the evaluation of both $h_1(u) \equiv h_1(g_1(x,y))$ and $h_2(u) \equiv h_2(g_1(x,y))$ as indicated in the definition of f_1 and is calculated only once.

By repeated application of the type of substitution shown above a single directed graph for each distinct process in an AIP process can be obtained. It is important to realize that

the semantic requirements of AIP specifications from this substitution at each step to yield an acyclic graph, namely in that "circular" definitions are not permitted. One final step is taken in order to simplify the algorithms and transformations which follow. It is simply to draw dummy arcs from a single vertex to each vertex labeled with state variables for a given process. This gives a graph which begins at a single vertex and ends at a single vertex. For the preceding example this step yields:



5.1.2 AIP Decomposition

Whenever a process from an AIP specification is defined in part by means of exchange functions then we can model the process by means of a single R-Net only where all exchanges are defined to occur exclusively at the beginning or end of a process step. (A single R-Net will also suffice wherever

a process is defined without exchange functions. Of course, in such instances the process in question does not interact with any other processes.) If an exchange takes place anywhere else within a process step, however, then we must use more than one R-Net so that computations both preceding and following exchange functions can be modelled. This is because an R-Net can receive a message only at its initiation point and can send messages only at its termination point(s). The actual transmission of messages and scheduling of R-Nets during a REVS simulation run is performed by Pascal routines, that is, non-RSL code. (The reasoning and motivation behind the previous statements as well as further details on REVS have already been given in section 5.0.)

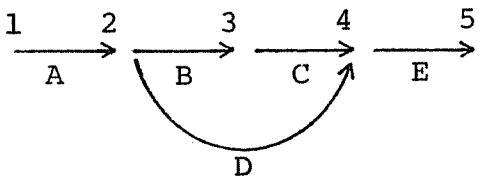
Out of the many possible ways to decompose state successor functions based upon occurrences of exchange functions in AIP specifications the particular method chosen for MRSL has been to minimize the number of R-Nets generated. This decomposition method has the result of simplifying the non-RSL code necessary for a simulation package by placing as much precedence and scheduling information in MRSL (versus Pascal) as possible. Since one of the primary purposes of RSL is in fact to represent flow-of-control information RSL seems like the most natural medium within REVS for expressing precedence information in the

---

[Fi78-3] Fitzwater, D. R. "The Formal Design and Analysis of Distributed Data-Processing Systems," Report DASG60-76-C-0080-3, August, 1978.

case of AIP. The first step in the decomposition is most easily presented as an algorithm whose input consists of the directed graphs (and associated vertex and arc labels) obtained in section 5.1.1 above. Two relationships among vertices and arcs which will be used extensively in the following material must be noted first, however.

We say that one arc precedes another arc if there exists a directed path from the first to the second. In this case we also say, alternatively, that the second follows the first. For example, in the graph:



arc A precedes arcs B, C, D, and E, arc B precedes C and E, and both C and D precede E. Also arcs B, C, D, and E follow A, and so on. Likewise, we say that vertex precedes an arc if the arc is incident on the vertex and is directed away from it or if the arc in question follows another such arc. For example, in the graph pictured above vertex 1 precedes all arcs (similarly all arcs follow vertex 1), vertex 2 precedes B, C, D, and E, and so on. Finally, we say that a vertex m precedes another vertex n if m precedes an arc that

precedes n. Since we are dealing only with acyclic directed graphs, a vertex or arc can either precede or follow another vertex or arc (but not do both), or do neither. Intuitively, the computations associated with non-dummy arcs can be performed only as soon as (1) the computations for all arcs preceding it have been performed, and (2) all exchange functions associated with vertices preceding it have been evaluated. We now present the decomposition algorithm (Algorithm 5.1) which partitions precedence graphs obtained from AIP specifications as described in section 5.1.1. Each of these (directed acyclic) subgraphs will eventually be transformed into an R-Net as explained in 5.1.3. The partition algorithm uses ordinary set notation and control structures familiar from a number of algorithmic languages. Comments are included with quotation marks. The algorithm is to be applied separately to each process in an AIP specification.

n: = 1 "n is the number of subgraphs currently identified by the algorithm"

R_1: = {a | a ϵ S_0, a precedes some $x \epsilon X_0$ and follows no $x \epsilon X_0$}

"R_i is the set of arcs in the i^{th} subgraph. So is the set of arcs in the entire graph for the process and X_0 is the set of vertices labeled with exchange function names in that graph.

Algorithm 5.1

```
X1: =    X0 - {x | x ∈ X0  and  x  immediately follows some
         member of  R1}
S1: =    S0 - R1
while  Sn ≠ φ  do
begin
     m: = n
     for  i: = 1  step 1 until  m  do
     begin
          if  Xn = φ  then
              T: = Sn
          else
              T: = {s | s ∈ Sn,  s follows some member of  Ri
                   but no member of  Xn,  and  s  precedes some
                   member of  Xn}
          if  T ≠ φ  then
          begin
              n: = n + 1
              Rn: = T
              Sn: = Sn-1 - Rr
              Xn: = Xn-1 - {x | x ∈ Xn-1  and  x  immediately
                   follows some member of  Rn}
          end
     end
end
```

Algorithm 5.1 (cont.).

It has been mentioned before that this algorithm decomposes the precedence graph for each process into the minimum number of subgraphs which can be modelled in RSL (with one R-Net per subgraph). These subgraphs consist of those arcs in sets R_i, $1 \leq i \leq n$, above and their associated vertices. Even though n represents a minimum, however, the algorithm above does not represent the only possible partition scheme for this minimum. From the point of view of REVS though any partitioning of the graph works as well as the others since each R-Net is considered to be evaluated instantaneously in simulated time. (Details on simulation will be given in section 5.3). Specifically in the algorithm above the inclusion of the phrase "and s precedes some member of X_n" in the last statement above where T is assigned a set value has the following consequence. Each R-Net will compute all that is logically possible from the point where exchange of a message or messages enables an R-Net to the point where all subsequent computation requires that one or more exchanges of messages be performed. The only exception is at the end of a process step where all computations have been completed and a new process step is ready to begin.

This completes the decomposition phase of the translation from AIP specifications into MRSL. The remaining transformations in the next section convert each subgraph obtained here into a R-Net structure. A detailed example which employs the transformations of this section and the next will be given in 5.2.

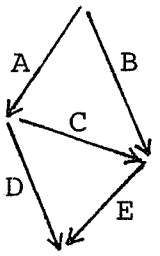
5.1.3 Modelling AIP with R-Nets

It has been shown in previous sections how AIP specifications can be reduced to precedence graphs which can be decomposed into subgraphs suitable for modelling in MRSL. In this section we will list five transformations necessary to convert each subgraph into an R-Net structure. An example illustrating these transformations will follow in section 5.2. In the following discussion, as before, graphs are described almost exclusively in terms of arcs rather than vertices since arcs represent computations and vertices are merely synchronization points. This lack of rigor permits us to highlight the main points of interest without obscuring them in unnecessary detail. The purpose of each transformation will be provided as an aid in understanding the overall scheme of converting AIP specifications to MRSL.

(1) In the first transformation a new directed acyclic graph is generated for each of the subgraphs R_i, $1 \leq i \leq n$, produced for some process by the algorithm of section 5.1.2 above. This is done in each case by deleting a different set of arcs and arc labels from the original precedence graph. The arcs deleted include the following for each i: (1) any arc that does not precede or follow an arc in R_i, and (2) any arc a_1 such that for some arcs a_2 and a_3, no two of the three arcs are in the same subgraph, a_1 follows a_2, a_2 follows a_3, and $a_3 \in R_i$. In addition all labels are removed from those remaining arcs and vertices which are not part of R_i. The only exception is that

the labels T and F on the dashed dummy arcs associated with selector functions remain. Note that the new graph G_i created in this way is a subgraph of the original precedence graph, and similarly R_i is a subgraph of G_i. The purpose of this transformation is to leave enough of the control structure "context" for each R_i intact so that the R-Nets for MRSL can be constructed. Actually the subgraph R_i alone contains enough information to generate a suitable structure for constructing an R-Net. However, the rules for doing so are more complicated than in the present approach. In many cases the dummy arcs produced for each G_i can be eliminated by means of transformation (3) below and simpler structures will result.

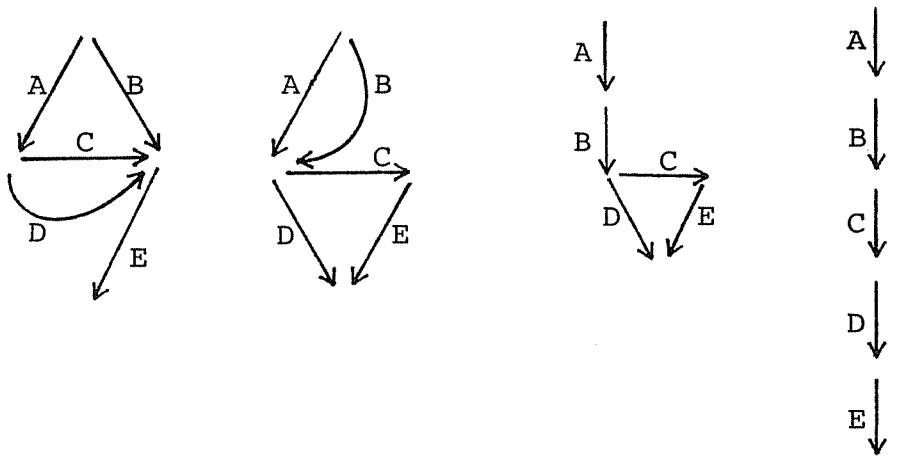
(2) A study of the finer points of RSL syntax and semantics is beyond the scope of this discussion. However, it is necessary to explore how precedence relations may be expressed in RSL because of the impact that this aspect of RSL has on MRSL, primarily in necessitating the inclusion of transformation (2), which will be presented below after a considerable digression into the motivation behind it. In AIP specifications we have permitted arbitrary precedence relations to be expressed, but unfortunately the block structure of RSL allows only a subset of these relations to be expressed. For example, in the simplest case we could have generated the precedence relation:



from some AIP specification, but this cannot be modelled directly in RSL. This problem has been solved in MRSL (at least insofar as the fidelity of the simulation model to be described in section 5.3 is concerned) by simply calculating the time necessary to simulate the computations of an R-Net based upon the original precedence constraints (as represented in each R_i) independent of the R-Net structure wherever the two conflict. The R-Net structure is then only an approximation of the original precedence constraints in such instances. It should be noted that these simulation times must be calculated in any case because R-Nets are considered within REVS to execute instantaneously in simulated time. The only disconcerting issue is that R-Net structures may be misleading if viewed independently from the precedence constraints which they attempt to model.

Transformation (2) will seem much clearer when we realize that the relationships "precede" and "follow" from the previous section induce a partial ordering on the computations modelled as arcs in a precedence graph. For our purposes any newly derived precedence graph which does not violate this partial ordering, that is, which may introduce new constraints without altering any old ones, will provide a suitable compromise

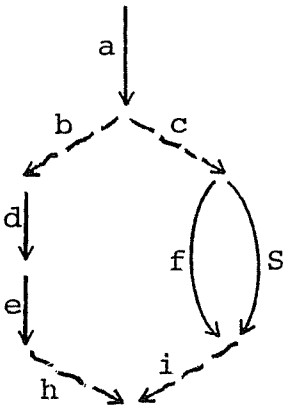
between precedence relations and RSL syntax requirements. Thus any of the following would be suitable transformations of the graph above:



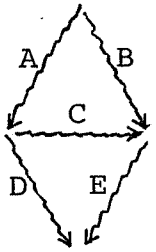
where the first corresponds to the algorithm to be given below and the last is a more extreme solution. (Incidentally, each of the four transformed graphs can be realized as an R-Net structure.)

Before giving transformation (2) we need one useful definition. Recall that each selector function from an AIP specification is translated into one or more nested quadruples of dashed arcs (see section 5.1.1). In the following algorithm (and also transformation (3)) an arc cluster is defined as the set of all arcs which follow one vertex and precede another under the following condition: if an arc cluster contains one dashed arc from a quadruple of such arcs than it must contain all four

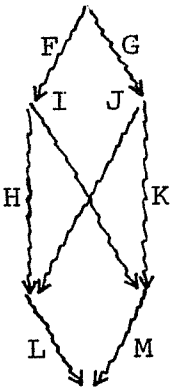
of them. For example in the following graph:



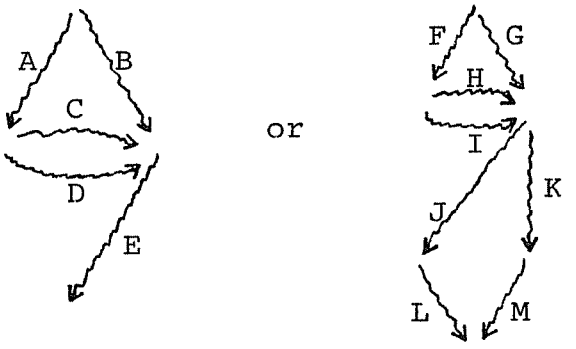
the sets {a}, {b}, {d}, {e}, {d,e}, {f,g}, {b,c,d,e,f,g,h,i}, and {a,b,c,d,e,f,g,h,i} are the only arc clusters. We will use a wavy arrow to depict each arc cluster in the algorithm for transformation (2) which we present at last: Repeat the following until the transformation can no longer be applied to the original or succeeding graphs. If a precedence graph contains (as a subgraph) either of the following configurations:



or



(where capital letters are labels of arc clusters for the purpose of identification in this algorithm only), then non-deterministically choose one such configuration and construct a new graph by changing the arc cluster configuration to:



as the case may be, and leaving all other precedence relationships unchanged and all labels unchanged.

(3) The third transformation is aimed at reducing the number of unnecessary dummy arcs resulting from transformations (1) and (2) above.

Perform the following two steps in either order to any portion of the (original or any succeeding) graph until neither is applicable anywhere in the graph.

(a) If an arc cluster consists entirely of two or more dummy arcs (including dashed arcs) then replace it with a single dummy arc. If this replacement results in the loss of one or more vertices labeled with an exchange

function name then label the vertex following the new dummy arc with those names in addition to any labels that the vertex may already have.

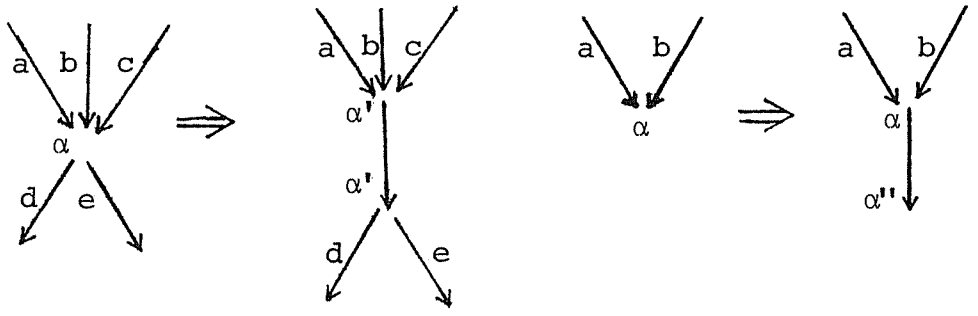
(b) If an arc cluster consists of two arcs where a labeled arc either precedes or follows a dummy arc, then replace it with the labeled arc alone. Retain vertex labels as in (a) above.

(c) If an arc cluster can be partitioned into two arc clusters where both have the same initiation and termination vertices and one cluster is a single dummy arc, then delete the dummy arc and leave the rest of the graph unchanged.

(4) This fourth transformation produces graphs which can be converted directly into R-Nets by transformation (5). The dummy arcs introduced here are essentially for the sake of uniformity of appearance in graphic R-Net representations since the arcs correspond to no syntactical construct in RSL source language.

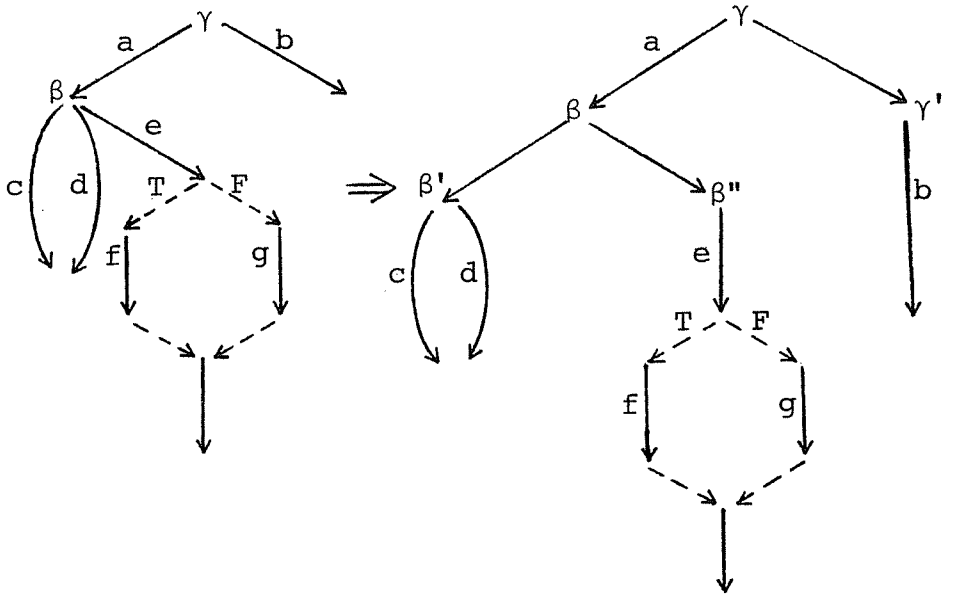
(a) Repeat the following until it is no longer applicable to the original or any succeeding graph: If a vertex α is preceded by zero arcs or more than one arc and is also followed by zero arcs or more than one arc then create a new graph identical to the first except that two new vertices α' and α'' with a dummy arc from α to α'' replace α such that all

(if any) arcs that preceded α now precede α' and all (if any) arcs that followed α now follow α''. Any vertex labels of α are retained by α''. Two pictorial examples help to illustrate this change:



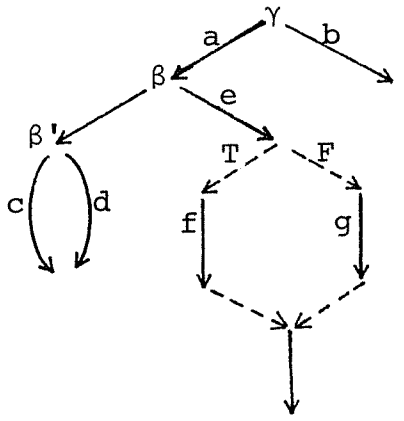
(b) Also insert a dummy arc at any vertex which is the initial vertex of a maximal terminal arc cluster unless there is only one such cluster (i.e., the entire graph). Retain vertex labels as in part (a). (A terminal arc cluster is any arc cluster having an arc followed by no other arc. In addition, it is maximal if it is not properly contained in any other terminal arc cluster.) If a vertex serves as the initial vertex for more than one maximal terminal arc cluster then a dummy arc is inserted for each cluster.

In the example below there are three such clusters, two at vertex β and one at vertex γ,



(c) Repeat transformation 3. (Some unnecessary dummy arcs may have been introduced at step (b) above and so are withdrawn immediately here.)

Taking the results of the example for step (b) above we obtain the following graph, where only one of the dummy arcs remains:



(5) The fifth and final transformation of this section converts the directed acyclic graph from step (4) above to an R-Net structure. We will be little concerned with labels on R-Nets since we have dealt mainly with control structures so far. The missing information would, of course, be presented in the RSL source language to which AIP specifications would be translated internally for the purposes of simulation. It should be noted, however, that R-Net drawings are an alternate and formally acceptable way to represent RSL control structures within the REVS methodology.

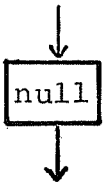
The following steps are to be applied in order once, where each succeeding step is applied to the result of the preceding step.

(a) If no arc precedes a vertex then designate it as an input interface (symbolically, a hexagon) plus an entry symbol (triangle, see later examples). (There is precisely one input interface per R-Net.)

(b) Similarly, if no arc follows a vertex then designate it as an output interface (also a hexagon). There can be one or more output interfaces per R-Net.

(c) If a vertex not already designated in (a) or (b) above is immediately preceded or immediately followed by a pair of dased arcs then designate it as an OR node (symbolically as a ⊕) otherwise as an AND node (symbolically as a Ⓐ).

(d) For any pair of dashed arcs (from the same quadruple of arcs) position the T arc and all arcs following it (but not following the F arc) to the left of the F arc and all arcs following it (but not following the T arc). By "left" we mean that traveling counterclockwise from any arc immediately preceding the pair we first encounter the T arc and then the F arc. Since all R-Nets are planar graphs this is always possible. In addition, if a single dummy arc follows the T arc or F arc and also immediately precedes one of the other dashed arcs in the same quadruple then replace it with the following configuration:

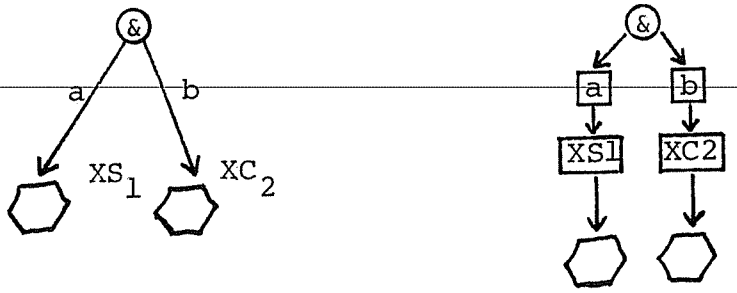


where the rectangle labeled "null" specifies that no computation takes place along that path. The rectangle (but not the "null" label) stands for an "alpha" in RSL. An alpha specifies a computation to be performed by a Pascal procedure. In the case of "null" alphas, which are inserted here only because of RSL syntax rules, the procedure is simply "BEGIN END".

(e) Let all dashed arcs "shrink" to a point, that is, merge the two vertices immediately preceding and following the dashed arc, thus eliminating the arc. The OR nodes above now serve the purpose of modelling selector functions from AIP.

(f) Replace each labeled arc by two arcs and an alpha with the same label, thus:

This means that a Pascal procedure will be used to compute each primitive function from the AIP specifications. If the labeled arc is not followed by any other arc (and hence precedes a vertex which is an output interface by (b) above) then include an additional alpha with a unique label preferably to include the names of all exchanges associated with or preceding the interface, as in the following example:



These alphas have the dual purpose of assigning state variables their newly computed values for the next system step (in those R-Nets which actually terminate a system step) and also for forming messages (for each exchange function named by vertices preceding the alpha) to be sent through their corresponding output interfaces. Thus an output interface may send more than one AIP message per RSL message. Any group of such messages, however, causes values obtained from evaluating these AIP exchange functions to be used only in a single R-Net, which in simulation runs will be executed subsequently. However, there is only one input interface per R-Net and so such an interface may receive exchange messages from one or more distinct R-Nets. It should be mentioned that additional information besides actual AIP message contents may be included in RSL messages. For example, execution times for various computation paths through the R-Nets may be included. Such topics will be treated in more detail in section 5.3. This transformation concludes the construction of R-Nets from AIP specifications. Section 5.2 will provide a detailed example showing the use of these transformations.

5.1.4 Data Flow in MRSL

Sections 5.1.1 to 5.1.3 have emphasized the representation of flow of control in MRSL models of AIP specifications and have dealt only peripherally with representations of AIP state spaces, state variables, evaluation of primitive functions, and treatment of messages within MRSL. Fortunately, these topics require far less detailed explanations than the MRSL control structures did. Much of this simplicity derives from the fact that AIP specifications are guaranteed to be complete and consistent, so that elaborate checks and cross references do not need to be made in translating from AIP specifications to MRSL. That is, we assume that the source code written in the syntax of section 3.4.3.2 of [Fi322] has been analyzed for adherence to the properties also listed in that section. (Other MRSL models will be discussed in section 5.3.5.). These properties guarantee that the source code actually constitutes a valid AIP specification.

Because of the many differences between MRSL and the AIP specification language some information which is explicit in AIP specifications (for example, state spaces, domains and ranges of functions, state successor functions, and some types of precedence information) is represented only implicitly in MRSL. Conversely, some information which must be an explicit part of MRSL is implicit in AIP specifications (for example, temporary variables, that is, for evaluating the state successor function, and the ranges of values which each state variable

may attain). It will become clear through later discussion and examples, however, that AIP specifications tend to be far more compact than corresponding MRSL specifications because the latter must be stated on the whole in a more semantically distributed and redundant way. For example, as we have already stated, every temporary variable in MRSL (including all variables used merely to send messages) must be declared individually as to its range of possible values, its locality of use, and its role in the data flow scheme of the R-Net in which it appears. All such information is implicit or meaningless in AIP specifications and temporary variables (which must in any case be created in order to simulate AIP specifications) are never mentioned. The remaining paragraphs of this section give the correspondences between AIP and MRSL language structures exclusive of flow-of-control structures.

State variables constitute the only type of variable in AIP specifications, but they represent one among several types of variables in MRSL. However they are readily distinguishable from other types in MRSL in that they are defined to be global (since they are shared among the R-Nets modelling the process of which they are a part, although they are not the only type of global variable) and further in that they are assigned values, by convention, only within the last alpha to be executed in a process step. They may be regarded as read-only variables during the initial portion of the process step. (Note that they are global variables only with respect to one particular process and

are never shared among processes.)

We have already mentioned that in AIP specifications the state space for each process is indicated explicitly, whereas in MRSL it is indicated only implicitly, namely by denoting the range of possible values which each state variable may take. In AIP specifications state variables may take values of type Boolean, integer, or string, and furthermore the range of values which they may take can be expressed in terms of unions of finite subsets of the above types. Since in MRSL we may declare variables of type "enumeration", thus in essence defining finite sets, we can represent all sets from AIP specifications by means of type "enumeration" in MRSL, where there exists a unique one-to-one correspondence between the members of the respective sets. It should be noted that sets in AIP specifications can also be defined in terms of cross products of sets, thus giving rise to n-tuples of variables. Although RSL has the capability of expressing hierarchical data structures, it gains us nothing to use this facility in MRSL to represent the tuples. This is because each state variable in MRSL must have a unique name and its mode of calculation must be given explicitly and individually. Thus the inclusion of a hierarchical ordering of MRSL variables is superfluous. In AIP specifications on the other hand a state variable does not have a specific name and need not be referenced uniquely or even explicitly. It is simply a component of the domain and range of some state successor function.

There are three remaining types of variables used in MRSL. The first of these comprises the global temporary variables. They consist precisely of all those global variables which are not state variables. Recall that state variables are assigned values only in the last alpha executed in a process step. Thus global temporaries may be assigned values anywhere else during the process step and are in fact identified as temporaries by being so assigned. The second type of variable not yet mentioned consists of those temporary variables which are locally defined (that is, created and used entirely within a single R-Net) but not used in any message. We will consider these temporaries in the same way as we do global temporaries in the following discussion because declaring them as global instead of local would have no effect on the MRSL specification. The third remaining type of variable consists of the local variables used to pass messages between R-Nets and drivers. These variables must by RSL (hence MRSL) semantic rules be declared as local. In section 5.2.2 we will provide an extended example which will illustrate the role of each type of variable in MRSL specifications. (Since this example was developed before all four MRSL models had been considered, its use of variables is not so strict as indicated in the scheme below which is applicable to each of the models. All deviations from this scheme will be noted.) Meanwhile we will give a brief outline of the uses of each of the respective types.

State variables in MRSL have already been described above. They correspond exactly to components in the domain and range of the state successor functions in AIP specifications. State variables in the example of section 5.2.2 are identified by their initial letters "SV" rather than by where they are assigned a value. On the other hand global temporaries represent all outputs of primitive functions modelled in MRSL. Each such output for any occurrence of a primitive function must have a unique name. Naturally some of these outputs are inputs to other primitive functions according to the precedence relations already established in section 5.1.1. (Other inputs to primitive functions are messages and state variables.) All global temporaries in the example of section 5.2.2 begin with the letters "IV". The third type of variable, namely local temporaries not used in messages, are used as necessary to store temporary results and timing information in the Pascal procedures that calculate primitive functions and also in those that form messages immediately before output interfaces. In the example of section 5.2.2 variables beginning with "LV", "EX", and "ELAP" are of this type as are variables beginning in "TIME" which are not used in messages. Finally, local temporaries used in messages merely provide buffer space (local to R-Nets) for messages passed between R-Nets and drivers. (Other message buffers to be explained in section 5.3.3 are provided separately by the Pascal simulation routines.)

In the example of section 5.2.2 these variables are listed for each message in the message definition statements of MRSL.

In the whole of section 5.1 we have dealt with those aspects of simulating AIP specifications which would be similarly represented in MRSL independent of which of the four models (given in section 5.0) is chosen for our AIP specifications. Those aspects which must be expressed in non-MRSL code or which are dependent on the choice of model are reserved for section 5.3. We will not describe the syntax of RSL here but instead refer the reader to [MD] for background information and section 5.2 of this report for examples. Since only a subset of the most transparent constructs of RSL is used in MRSL, the examples of this report will be virtually self-explanatory. The only exception is for R-Net structures, which we have expressed so far only in graphical terms. In the example of section 5.2.2 the textual representation of the R-Net structures derived in section 5.2.1 are given without explanation. The formal correspondences between the two representations are given in section 3.3 of [MD].

We now list all of the language constructs of MRSL which must be included in a valid specification. In MRSL information is expressed by means of what can be considered a declaration plus its corresponding definitions. We present the possible

declaration-plus-definition types in the order in which they occur in the example of section 5.2.2. This order is completely arbitrary and of no special significance. There is enormous flexibility within MRSL for reordering and restructuring lists of definitions without changing the overall MRSL specification in any respect.

(1) Input interfaces and output interfaces must be declared and defined in MRSL. They have been mentioned already in connection with R-Net structures. Input interfaces receive messages from the exogenous event routine during a simulation run, and the associated R-Net is considered to be entirely executed at the simulation time when such a message is received. (See [MD] for details. In RSL messages may be sent to input interfaces from any Pascal driver procedure, not merely the exogenous event routine.) In RSL terms the input interface is said to "enable" its associated R-Net, and in MRSL in particular every R-Net has such an interface. The MRSL message contains some simulation timing information and possibly one or more AIP messages (that is, exchange function arguments). Similarly, output interfaces pass MRSL messages to simulation drivers and thus schedule them for execution. An R-Net in MRSL may have one or more (in RSL, zero or more) output interfaces. Each output interface corresponds to passage of control from the current R-Net to one other R-Net (possibly itself) as determined by the decomposition algorithm of section 5.1.2. In addition, of course, the output

interface also passes one or more AIP messages (except possibly at the end of a system step) via a single MRSL message, which also transmits some simulation timing information (to be explained in section 5.3.5). In conclusion, the declaration of an output interface has the purpose of relating an R-Net and a unique message name and a unique driver name.

(2) Declarations of R-Net structures come next in the example of section 5.2.2. As we have already mentioned these correspond to the graphic examples of section 5.2.1. The correspondences between the two representations can be found in section 3.3 of [MD] and will not be discussed further.

(3) Each message passed by the input and output interfaces described in (1) above must have its constituent variables listed as part of its definition. These variables must likewise be declared as local to the R-Net which receives or sends the messages. In MRSL each message corresponds to precisely one interface, and further (unlike RSL) each interface corresponds to precisely one MRSL message definition. The distinction has already been repeatedly made between an AIP message (originating from an exchange function) and an MRSL message which may contain zero or more AIP messages plus some additional information depending upon the choice of AIP model (see sections 5.3.3 and 5.3.5).

The particular exchange functions involved in each MRSL message can easily be obtained from decomposition algorithm 5.1 in section 5.1.2. The exchanges for all the output interfaces

for the ith R-Net in a process are contained in the set $X'_i = X_{i-1} - X_i$, where X_{i-1} and X_i are obtained from the algorithm. We partition X'_i into subsets so that each subset is the set of all exchanges immediately preceding arcs in R_j for some j, where all R_j are likewise obtained from the algorithm. The exchanges represented by each such subset contribute to an individual MRSL message. (At the end of a system step the X'_i is not partitioned.) It should be noted that because of selector functions not every AIP message will be sent by its containing MRSL message on every system step. This eventuality must be included as part of the message. In addition, the time that the AIP message becomes pending must be included in the MRSL message so that it can be placed on the simulation event calendar. The above information is the minimum that can be placed in an MRSL message. In some AIP models additional information is included. Section 5.3.5 expands on these topics.

(4) The next set of definitions which must be included in any MRSL specification is for alphas. So far, alphas have been discussed principally as the type of node within R-Nets which is associated with computation of primitive functions and not merely with flow of control. Additional alphas used immediately before output interfaces are for the utilitarian purpose of (1) calculating timings for use in scheduling exchanges and R-Net executions on the simulation event calendar, (2) forming MRSL messages for the output interface, and at the end of a system step, (3) assigning newly calculated values

to the state variables with which the containing R-Net is associated. The methods by which alphas calculate simulation timing values is straightforward and will be given in section 5.3.2. However, the fact that an alpha forms a message is simply stated as part of its definition and does not require any other MRSL code since REVS instead generates the necessary Pascal code automatically.

All computation within an alpha (of either the primitive function type or the output type) is specified by a single Pascal procedure. This procedure, given within double quotes in MRSL, lacks arguments and a heading. The heading is instead supplied by REVS when the procedure is inserted into the Pascal simulation program. In addition, REVS also inserts a procedure call within this code if the alpha forms a message. Finally, it is necessary to declare all variables used as inputs to the procedure (that is, previously assigned variables) and all variables which are outputs of the procedure (that is, variables assigned a value within the procedure) as shown by examples in section 5.2.2.

(5) Finally, all data used in the MRSL specification must be declared as to type, locality, and (in some cases) initial value. The data will consist precisely of all the variables listed as inputs and/or outputs in (4) above. The attributes of type and locality have already been discussed at length. The initial value of a variable is required for state variables in order to define the system state at the beginning of a simulation.

5.2 MRSL Examples

5.2.1 R-Net Generation

Many of the graph operations and characterizations given in section 5.1 were introduced without examples in anticipation of the more unified presentation of this section. Here the precedence graph for the state successor function for one process specification will be carried through the entire set of operations of sections 5.1.2 (decomposition) and 5.1.3 (MRSL structures). For another process which interacts with the first via exchanges we will provide only the final results of applying all the transformations in order. (Since this second process is much simpler than the first, very little can be gained by laboriously stepping through all the transformations after it has been done already three times for the first process.) Recall that the final graphs obtained are mere flow-of-control abstractions of true AIP specifications. (The missing information includes the state variables and state space for each process as well as the procedure for computing each primitive function and assigning a simulation timing value to each computation. None of this information is relevant to the current discussion and so is postponed until section 5.3.) The two example processes have, on the other hand, been tested in their completely specified final form by means of simulation runs with existing REVS software. These examples were prepared for testing entirely by hand, although the

algorithms of section 5.1 are quite straightforward and could be easily automated.

We begin our examples with precedence graphs which are assumed to have been already derived from AIP specifications as shown in section 5.1.1 (which already has examples for all the graph transformations contained therein). Thus only the information required for the operations of decomposition and subsequent transformations has been extracted from the original AIP specifications. For the purposes of the decomposition algorithm of section 5.1.2 the dummy arcs in the graphs below have been labeled D1,D2,...,D18 whereas non-dummy arcs are labeled A1,A2,...,A17. The labels for the dummy arcs will be dropped for all subsequent transformations in order to avoid confusion.

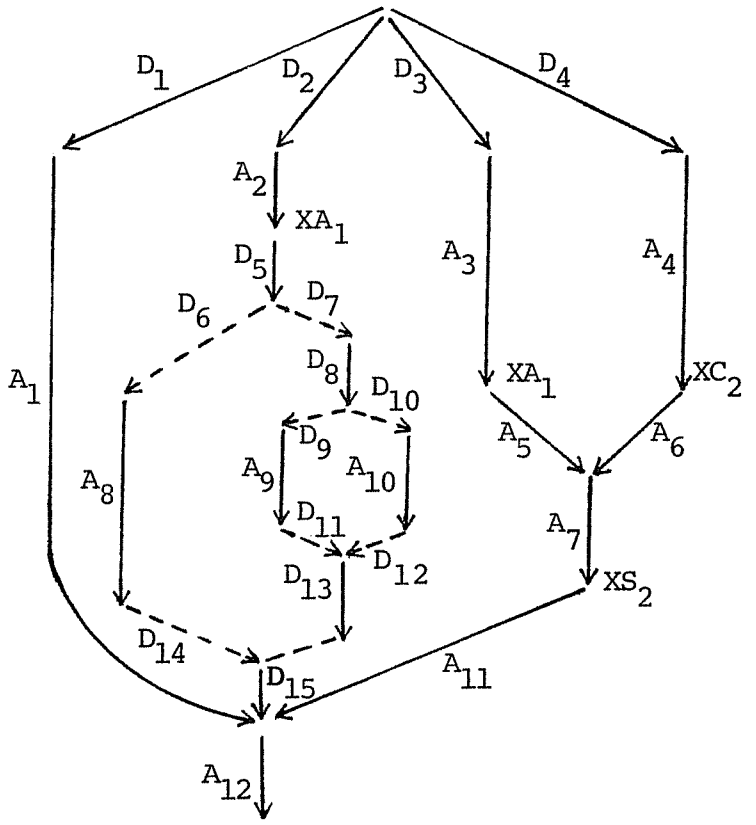


Figure 5.2.1. Precedence Graph for Process 1.

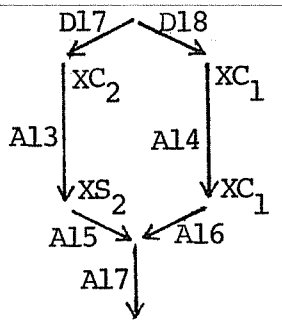


Figure 5.2.2. Precedence Graph for Process 2.

We now give a complete trace through the algorithm of section 5.1.2 for process 1 above. S_0 is the set of all arcs {A1,A2,...,A12,D1,D2,...,D16}, and X_0 is the set of all vertices labeled with the name of an exchange function, hence $\{XA_1(\text{after A2}), XA_1(\text{after A3}), XC_2, XS_2\}$. In the trace below, logical decisions which dictate the choice of the next instruction are given in parentheses.

$S_0 \leftarrow$ {A1,A2,...,A12, D1,D2,...,D16}

$X_0 \leftarrow \{XA_1(A2), XA(A3), XC_2, XS_2\}$

$n \leftarrow 1$

$R_1 \leftarrow$ {A1,A2,A3,A4,D1,D2,D3,D4}

$X_1 \leftarrow \{XS_2\}$

$S_1 \leftarrow$ {A5,A6,...,A12,D5,D6,...D16}

$m \leftarrow 1,\ i \leftarrow 1$

$(X_1 \neq \emptyset)$ $T \leftarrow$ {A5,A6,A7}

$(T \neq \emptyset) n \leftarrow 2$

$R_2 \leftarrow$ {A5,A6,A7}

$S_2 \leftarrow$ {A8,A9,...,A12 D5,D6,...,D16}

$X_2 \leftarrow \emptyset$

$(S_2 \neq \emptyset) m \leftarrow 2,\ i \leftarrow 1$

$(X_2 = \emptyset)$

$T \leftarrow$ {A8,A9,...,A12, D5,D6,...,D16}

$(T \neq \emptyset) n \leftarrow 3$

$R_3 \leftarrow$ {A8,A9,...,A12, D5,D6,...,D16}

$S_3 \leftarrow \emptyset$

$X_3 \leftarrow \emptyset$

$m \leftarrow 2$

$T \leftarrow \emptyset$

$(T = \emptyset, S_3 = \emptyset,$ algorithm halts)

Figure 5.2.3 Trace of Algorithm 5.1 for Process 1.

The results of the application of algorithm 5.1 to process 1 are that three subsets of arcs have been identified for eventual transformation into R-Nets, namely:

R_1 = {A1,A2,A3,A4,D1,D2,D3,D4},

R_2 = {A5,A6,A7},

and R_3 = {A8,A9,...,A12,D5,D6,...,D16}.

Likewise, if the algorithm is applied to the precedence graph for process 2 then we obtain the three subsets:

R_1 = {D17,D18},

R_2 = {A13,A14},

and R_3 = {A15,A16,A17}.

The remainder of this section illustrates the graph transformations of section 5.1.3 almost entirely by means of figures. The operations will not be repeated in words here and so several references to section 5.1.3 will be necessary for an understanding of each figure. In each figure we name the transformation(s) which transform one graph to another graph over a double arrow. If we place a number n in parentheses following a transformation or step of a transformation it means that the transformation has been applied n times. Also if a transformation is omitted in a sequence, for example, if we apply transformations 1 and 3 but not 2, then the omitted transformation is not applicable to the

graph. Another way of saying the same thing is that the transformation omitted has no effect or is a null transformation. The same explanation holds when a transformation contains several steps, that is, if only transformation steps 3a and 3c are shown (steps a and c of transformation 3) then step b is not applicable.

For process 1 we have already shown that the decomposition algorithm yields three subsets of arcs, each of which will eventually yield an R-Net structure. Each one departs from the graph of figure 5.2.1, which will not be redrawn. We name the subsets of arcs and the final R-Nets for process 1 with the letters A, B, and C, respectively. Without further explanation we present the figures illustrating the transformations of section 5.1.3 on each of the subsets of arcs A (Fig. 5.2.4), B (Fig. 5.2.5), and C (Fig. 5.2.6).

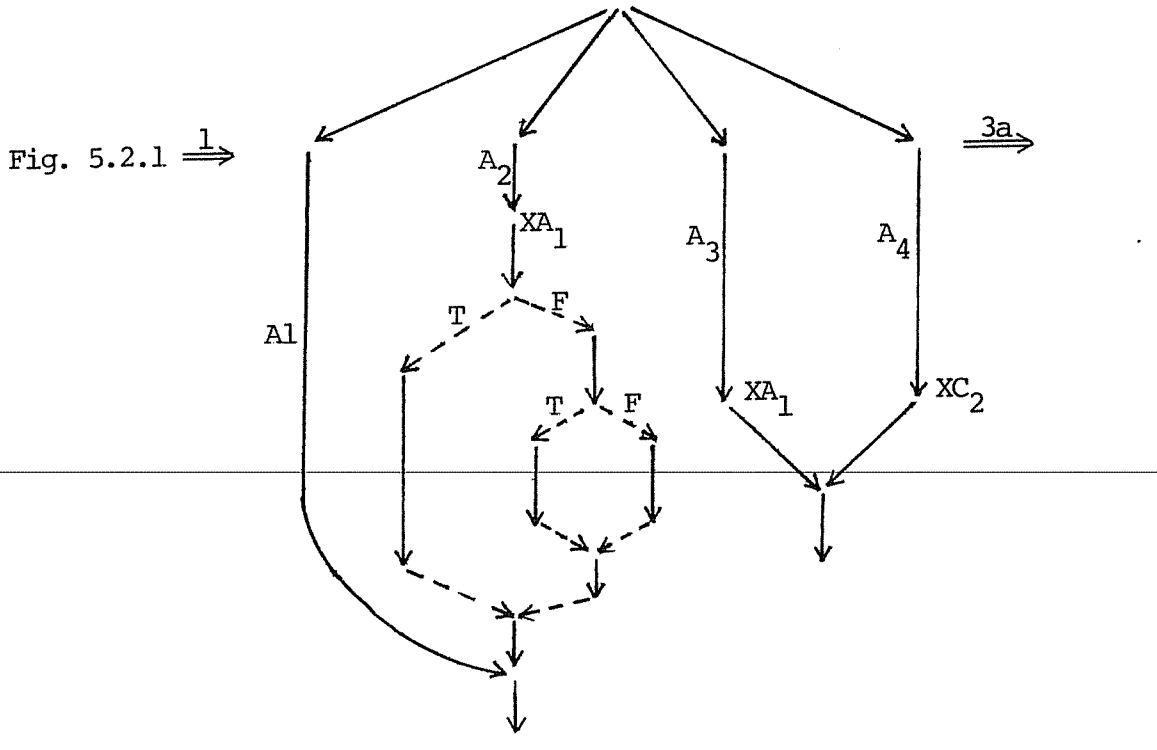


Figure 5.2.4. Generation of R-Net A.

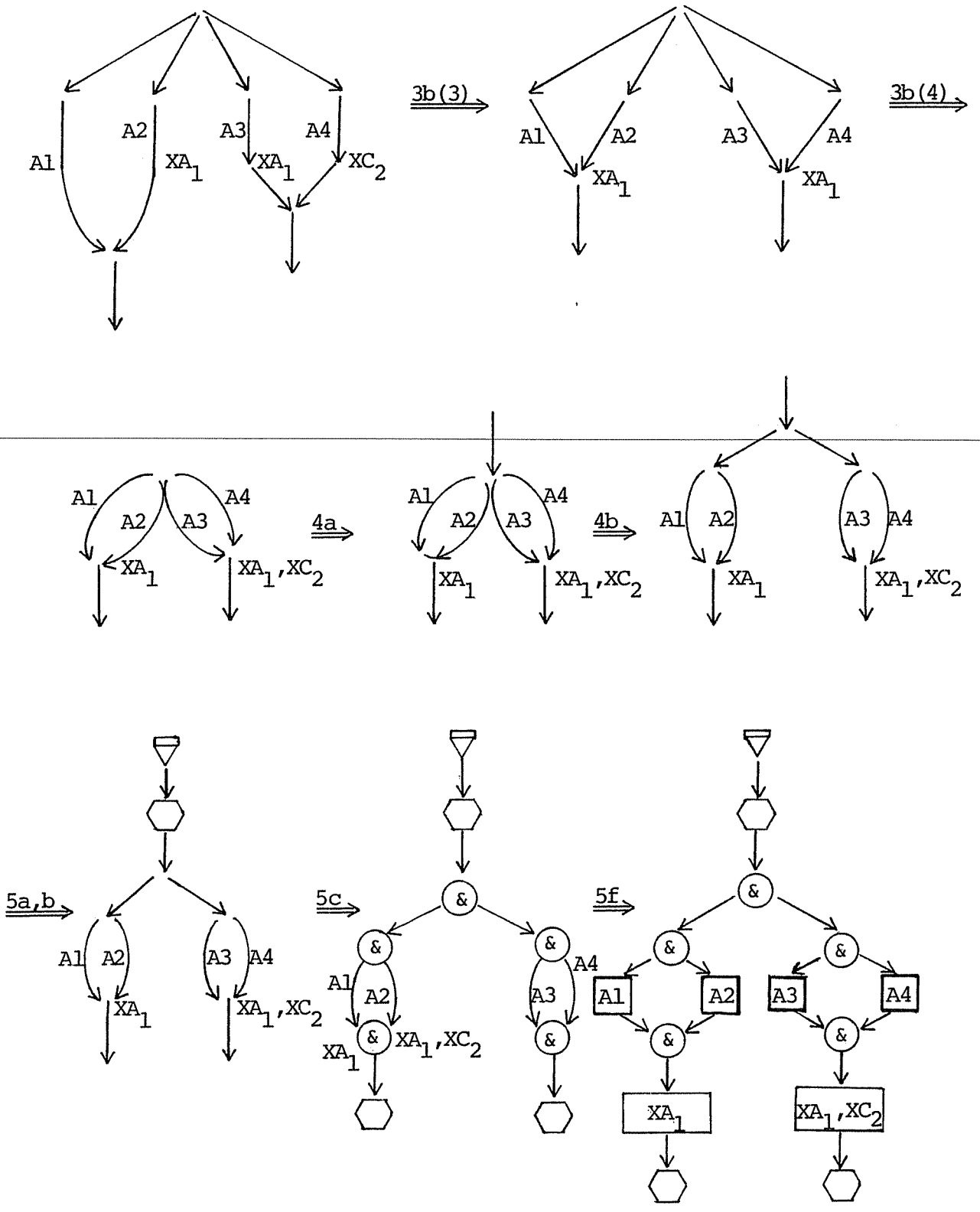


Figure 5.2.4(cont.) Generation of R-Net A.

Fig. 5.2.1

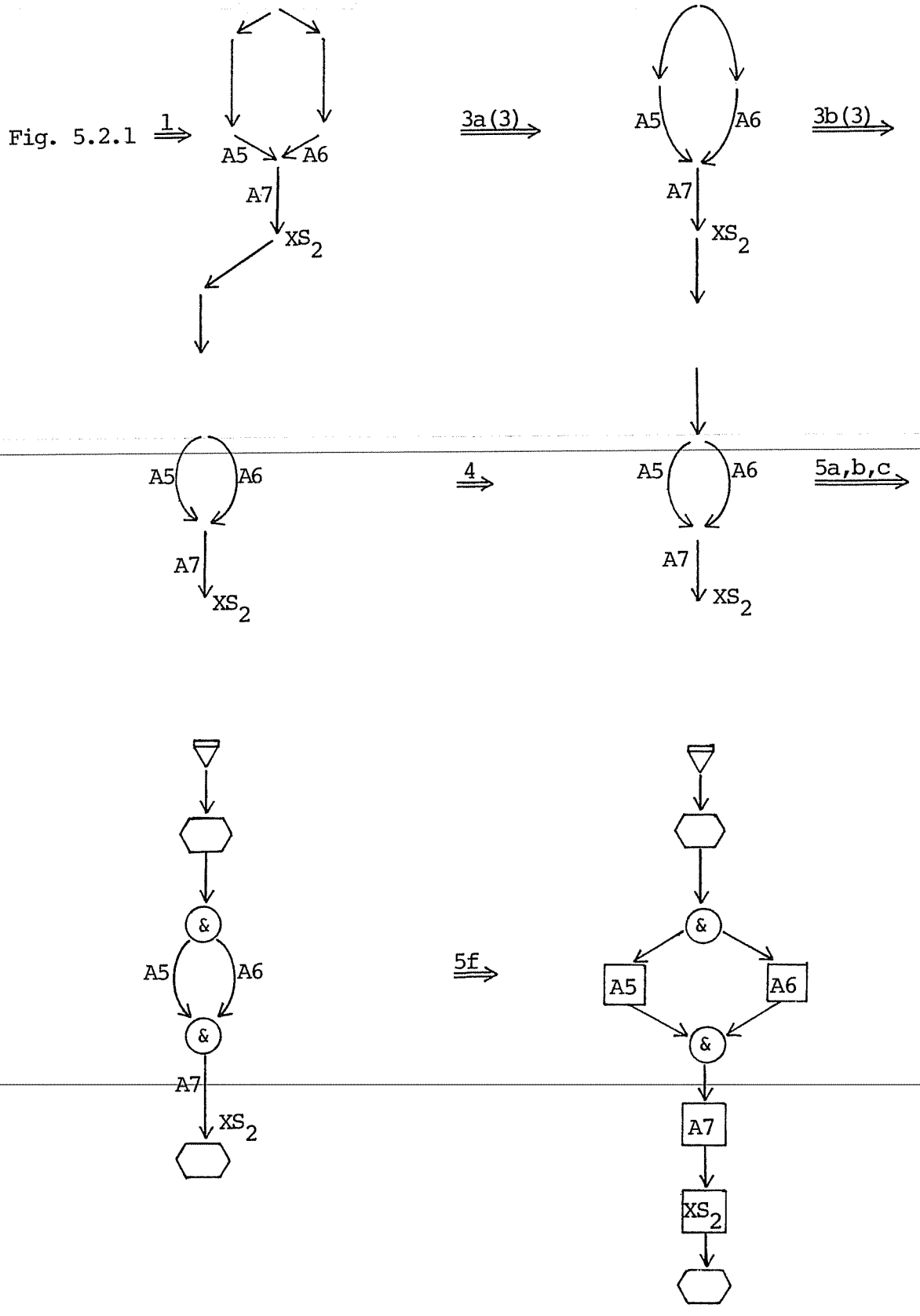


Figure 5.2.5. Generation of R-Net B.

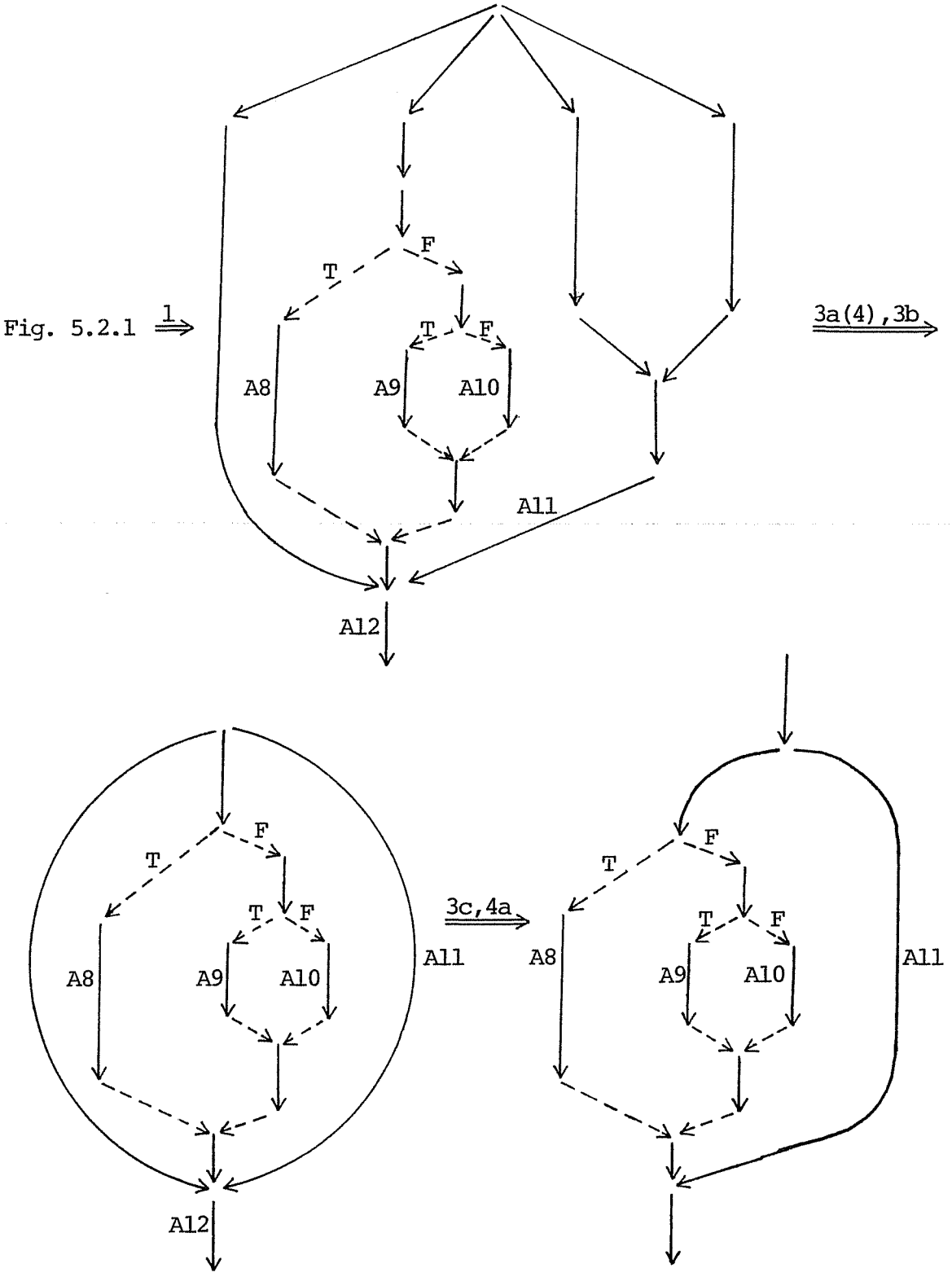


Figure 5.2.6. Generation of R-Net C.

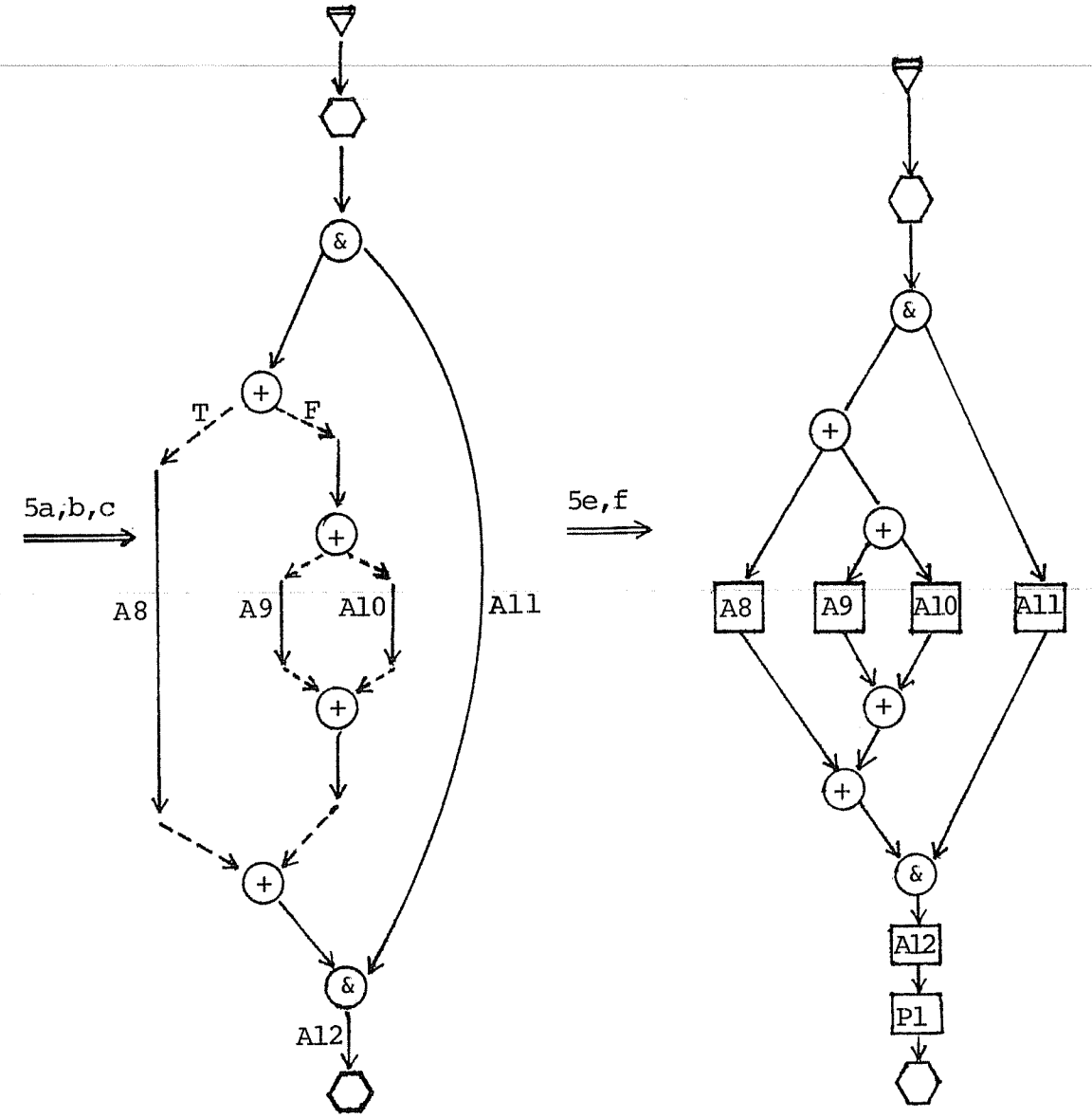


Figure 5.2.6(cont.) Generation of R-Net C.

We now give two more R-Nets derived from process 2 (Fig. 5.2.2). Note that although three subsets of arcs are obtained for process 2 via the decomposition algorithm, the first contains only dummy arcs. Instead of creating a dummy R-Net for this subset we simply allow it be be absorbed into the simulation routines described in section 5.3. We label the two R-Nets obtained from process 2 as R-Net D (Fig. 5.2.7) and R-Net E (Fig. 5.2.8).

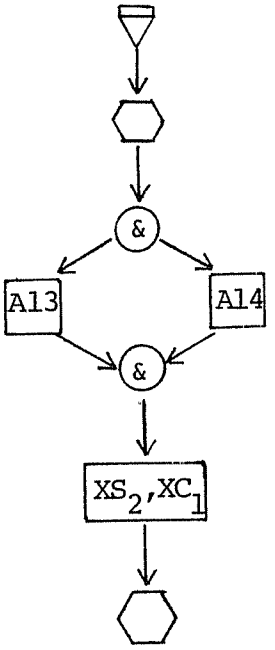


Figure 5.2.7. R-Net D for Process 2.

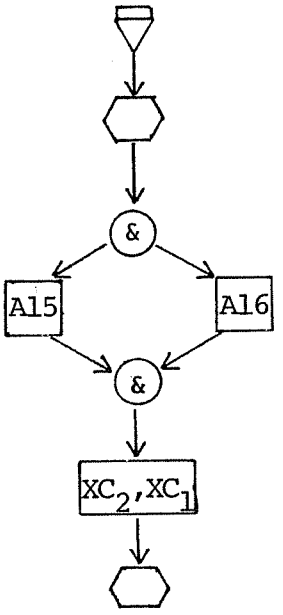


Figure 5.2.8. R-Net E for Process 2.

5.2.2 MRSL Specifications

Many references have already been made to the example here in section 5.1.4. Also the code was explained there to a level where no further discussion is necessary. Thus we end the section by referring to the listing A.1 of Appendix A.

5.3 MRSL Simulation

5.3.1 Overview

In sections 5.1 and 5.2 we have discussed MRSL in isolation from the remainder of REVS. Recall that MRSL is a subset of RSL in three interpretations of MRSL (the AIP model, the table model, and the message model) and is a superset of RSL in one other interpretation, namely the augmented RSL model, in which the language is expanded in order to express relationships and constructs necessary for AIP but unavailable in RSL. In none of the models above can a REVS simulation, based upon MRSL code alone, be carried out (via SIMGEN). In every case, as we have indicated previously, it is necessary to supply Pascal routines to drive the simulation by passing AIP messages and by placing items on the event simulation calendar. In the case of the message model and the augmented MRSL model these routines can be supplied automatically from an analysis of the MRSL specifications. However, in the AIP model and the table model the MRSL specifications are incomplete because the accompanying Pascal routines for the simulation, although standardized, must be parameterized in order to express relationships among R-Nets and messages which form an integral part of the overall AIP specification.

It will be seen in section 5.3.5 that the differences among the four models are not great; they merely represent

alternative notational conventions. We continue to pursue primarily the AIP model in this section, providing in section 5.3.3 examples of the Pascal simulation routines and in section 5.3.4 some sample output from an actual simulation run. The other MRSL interpretations are then compared in section 5.3.5.

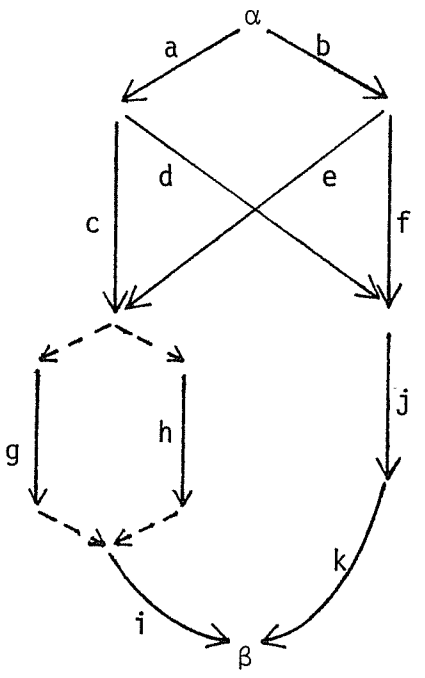
5.3.2 Time in REVS Simulations of MRSL Specifications

Before we can introduce the Pascal drivers for inclusion in a REVS simulation program for a given set of MRSL specifications we need to discuss the role of simulation time in MRSL itself, a subject only briefly mentioned in section 5.1.4. We assume that we begin with a precedence graph of the type obtained in section 5.1.3 after the first transformation. We further assume that every primitive function requires a finite amount of simulated time to produce an output once all of its inputs have been calculated. This elapsed simulation time in general may be considered as a function of the input values just as in complexity theory. However, the internal calculations of the primitive functions and the times necessary to perform such calculations are beyond the scope of our discussion.

Given the fact that each primitive function requires a finite amount of time for evaluation, we can easily calculate the time necessary to traverse any computation path within the precedence graph. Here we consider a computation path between two vertices to consist of all arcs which fall on any

two vertices along the path. We take the calculation time between them to be the maximum obtained for any of the parallel branches. On the other hand wherever computations are constrained to be performed serially the computation times for the individual subpaths are simply summed. (The two schemes for parallel and serial computation apply recursively to the precedence graph, of course, as we shall see from the more formal presentation below.)

The ideas of the preceding paragraph can be formalized as follows. We consider dummy arcs which do not immediately follow a vertex labeled with an exchange function name to have an execution time of zero. For those dummy arcs appearing immediately after an exchange label we consider the time necessary to perform the exchange to be the "computation" times of the arcs. In addition, all arcs along a branch of a selector function which is not selected are never evaluated and so are considered to have an execution time of zero without exception. Intuitively then for a particular choice of state variables and temporary variables the execution time for a computation path (defined between two vertices) is the maximum value obtained for any directed path (between the two vertices) by summing the computation times for the arcs on that path. For example, consider the precedence graph below where the arc labels represent computation times.



The computation time for the computation path defined by vertices α and β is:

$$t(\alpha,\beta) = \max\{(a+c+g+i),\ (a+c+h+i),\ (b+e+g+i),$$

$$(b+e+h+i),\ (a+e+j+k),\ (b+f+j+k)\},$$

where each sum represents a directed path from α to β.

Actually, it is not necessary to consider each directed path in the precedence graph for any particular state successfor function. If we consider only those computations paths which proceed from the initial vertex of the state successor function then a simple induction argument gives a more useful result. We can in fact compute for any arbitrary

vertex β the time corresponding to the computation path from the unique initial vertex α to the vertex β simply by considering (1) the times associated with those vertices immediately preceding β (which we will assume by induction to have been already calculated) and likewise (2) the computation times for the arcs immediately preceding β. Formally we define a new function T such that $T(\alpha)$ is assigned the time value τ for the beginning of the system step corresponding to the precedence graph of which α is the initial vertex. We define T for other vertices as follows. Let $\beta_1,\beta_2,\ldots,\beta_n$ be those vertices immediately preceding some arbitrary vertex β. Then let

$$T(\beta) = t(\alpha,\beta) + \tau = \max\{t(\alpha,\beta_1)+t(\beta_1,\beta),t(\alpha,\beta_2)+t(\beta_2,\beta),$$

$$\ldots,t(\alpha,\beta_n)+t(\beta_n,\beta)\}$$

Note that $t(\beta_i,\beta)$ is for $1 \leq i \leq n$ simply the computation time for a single arc. Also by calculating values of T in any order which preserves the partial ordering induced by the precedence relations inherent in the graph each $t(\alpha,\beta_i) = T(\beta_i)$ in the above equation will have been calculated previous to the calculation of $T(\beta)$. Thus we need compute $T(\beta)$ only once for each vertex β and the calculation itself is almost trivial.

This latter formulation for calculating computation times is very useful after the partitioning of the precedence graph for the state successor function has been accomplished

by means of algorithm 5.1. In general, several subgraphs will result from the decomposition. For the subgraph containing the initial vertex α we need only know $T(\alpha)$ in order to compute time values for the other vertices in the subgraph. (When a vertex is both preceded and followed immediately by arcs of two different subgraphs then it is considered to be part of the following subgraph.) For all other subgraphs we need to know the values of T only for those vertices immediately preceded by arcs in some other subgraph and further we need the total simulation time elapsed during the exchange of messages whose labels fall on vertices in the subgraphs in question. (The latter information, just like $T(\alpha)$, is supplied by simulation routines and is not calculated internally by code reserved for modelling computations of which precedence graphs are abstractions. More details on simulator routines follow in section 5.3.3.) For example if γ is one such vertex in some subgraph V then let $B_1, B_2, \ldots, B_m$ be the arcs not in V which immediately precede γ, let $b_1, b_2, \ldots b_m$ be their respective computation times, and let $\beta_1, \beta_2, \ldots \beta_m$ be the respective vertices which immediately precede them. For those arcs $C_1, C_2, \ldots C_n$ which are in V and immediately precede γ (if they exist) let $c_1, c_2, \ldots c_n$ be the respective computation times and $\gamma_1, \gamma_2, \ldots \gamma_n$ be the respective vertices. Also let

$$T'(\gamma) = \max\{T(\beta_1)+b_1, T(\beta_2)+b_2, \ldots, T(\beta_m)+b_m\}$$

and $T''(\gamma) = \max\{T(\gamma_1)+c_1, T(\gamma_2)+c_2, \ldots, T(\gamma_n)+c_n\}$,
where $T''(\gamma) = 0$ if no arcs C_i exist. By the definition of T we have immediately that

$$T(\gamma) = \max\{T'(\gamma), T''(\gamma)\}.$$

Thus the only information on simulation times which is not local to a subgraph consists of $T'(\gamma)$ for every γ in V which has one or more vertices outside V preceding it and also those times denoting receipt of AIP exchange messages. Since R-Nets correspond one-to-one with the subgraphs under discussion we have the result that the timing information just mentioned is all that needs to be saved between R-Net executions during a REVS simulation run.

We have described computation times so far only in terms of the original precedence graphs for a state successor function rather than in terms of the R-Nets generated from these graphs. This is done out of necessity because the precedence graphs can express precedence information with greater generality than can the R-Nets, as we have mentioned previously in connection with transformation (2) in section 5.1.3. In other words we must capture the correct precedence relationships with respect to simulation timings for the modelling of AIP computations before those precedence relationships are possibly altered during R-Nets generation. In RSL terms the correct computation times are first supplied individually by each alpha (which corresponds to a labelled

arc in the precedence graph) and then the alphas inserted before each output interface collectively calculate T' values for succeeding R-Nets. These values are then passed between R-Nets via MRSL messages which also contain true AIP exchange messages. However, the simulation timing values are not "exchanged" as such but instead are used by simulation drivers for scheduling purposes and then passed to the appropriate R-Nets. More detail on the role of computation times will be provided in the following section on simulation routines.

5.3.3 Pascal Simulation Routines

All the routines which must be included within REVS in order to run a simulation of MRSL specifications must be written in Pascal. The SIMGEN function of REVS receives both the MRSL code and the Pascal code, then translates the MRSL code into Pascal code, and finally inserts all of the application-dependent code into a standard host Pascal program. This host program is equipped with a large number of procedures for dealing with RSL constructs and also with simulation tasks which may be called from non-host code. Actually MRSL does not make use of all these standardized routines, and furthermore the Pascal routines are themselves in standardized forms which (in the AIP model) are essentially parameterized in order to encode interrelationships concerning R-Nets which are not expressible in MRSL.

These standardized procedures will be discussed below with reference to the data structures they manipulate. Example code is located for reference in Appendix A, section A.2.

The simplest of these standardized procedures is for the purpose of initializing the simulation. Its name, SSSTARTUP, is fixed by the conventions of REVS. In the AIP model for MRSL specifications its only purpose is to activate the first R-Nets to execute in the simulation, that is, those corresponding to the beginning of a system step. This is done by sending an MRSL message to each such R-Net with the simulation timing variables set to the appropriate values. In our examples (Appendix A, sections A.2 and A.3) the initiation time is zero. (The simulation timing variables used for initiating the simulation are precisely those variables which take values corresponding to the $T(\alpha)$ of section 5.3.2. This holds true for any of the four MRSL models.) The initialization procedure also sets the simulation tables so that they indicate that no R-Nets have yet executed and that no AIP messages are pending. (These tables will be discussed more fully below.) Recall that the values of the state variables of AIP specifications are initialized in MRSL itself and so are not involved with the SSSTARTUP routine.

The next group of standardized simulation procedures is associated with output interfaces and so according to REVS

terminology they are called drivers. There is one driver per output interface and furthermore each driver is scheduled for execution whenever its corresponding output interface passes an MRSL message. Since, as we have already noted, R-Nets are executed instantaneously with respect to simulated time, each MRSL message contains zero (only at the end of a system step) or more AIP messages along with the simulation times at which they will be considered to have become pending. These time values are simply T'(α) values (see the preceding section) which happen to be associated with message-labeled vertices. Additional T'(α) values may be contained in an MRSL message, of course, depending on the R-Net structure.

The function of the drivers is to place the AIP messages and timings in the appropriate simulation tables for future reference by the exogenous event routine (to be explained shortly) which serves in MRSL principally as a scheduler and message router. In the example code of Appendix A, section A.2, the drivers (procedures SAE1_2 through SEE) accomplish this by calling two very short preceding routines (1) MESBUF, which merely places the contents of each AIP message in a reserved buffer location along with the type of exchange function from which the message originates and the exchange class, and (2) CAUSE, which takes note (in a table of parameterized size, cf. SSSTARTUP) of the fact that the AIP message will be considered at some future simulation time and schedules an event on the internal REVS simulation calendar

for that purpose. Finally, if an AIP message is absent from an MRSL message because its originating exchange function has not been evaluated in an R-Net, then special note is taken of this fact and no corresponding event is scheduled. (Note that in the drivers of section A.2 all assignment statements are vestigial remains of earlier types of simulation runs. They could be removed here without effect since the exogenous event routine nullifies their action.)

The last and most complex of the standardized simulation procedures is called the exogenous event routine (code name SSEXOG) by REVS convention. (It is special in REVS in that its execution is event driven and not the direct result of the passing of any RSL message.) This routine is always activated in MRSL as the result of the scheduling of an AIP message for exchange or as the result of the termination of a system step for some process. It is perhaps best to explain the major steps that this routine performs by means of an informal algorithm owing to the opacity of the Pascal code in section A.2. This algorithm is performed each time an event on the REVS calendar activates the exogenous event routine and steps (1)-(5) below are executed sequentially, of course.

(1) Locate an event scheduled to be examined at the current simulation time and flag it as having been so selected. (This is necessary because two or more events may be scheduled simultaneously, and each must be treated individually).

(2) If no messages are associated with the selected event (i.e., in case of the termination of a system step) then schedule the next R-Net immediately and skip the remaining steps (3)-(5).

(3) If the message (event) selected in (1) above can be found to match another message of the same class which became pending previously and the match is of type XC - XC, XC - XA, XS - XC, or XS - XA then select the earliest such message, exchange the contents of the two message buffers, skip step (4), and perform step (5) twice, once for each exchange message. Messages which have been matched are then excluded from any further consideration and the buffer space is freed for future messages.

(4) If the message selected in (1) is of type XS then it is allowed to perform a self-exchange. It is removed from further consideration, its corresponding buffer space is freed, and step (5) below is executed once. On the other hand, if the message is of type XC or XA then the algorithm terminates with this step.

(5) Place the contents of the message from step (3) or (4) in the buffer for the input interface of the R-Net to which the message is to be dispatched. The current simulation clock time is the correct T' value associated with that message and should also be placed in the buffer. If the buffer is full then activate the R-Net for execution immediately by "posting" the MRSL message for the R-Net's input interface and then free the buffer for the next system step.

5.3.4 Example Simulation Run

The entire Pascal program for simulating the sample MRSL specifications derived in section 5.2.1 takes up over 2800 lines of code, only a fraction of which are specific to the current example. Most of the latter are in section A.2 or else consist of the Pascal code (for alphas) embedded in the MRSL code shown in section A.1 of Appendix A. We do not reproduce the entire listing of the simulation program generated by REVS here, but we do provide the first 250 or so lines of output for a run of that program. This output contains rather readable REVS-generated output which is almost obscured by lists of unlabeled numbers generated by the simulation routines in their debugging phase. Each block of eight rows of numbers and Boolean values corresponds to one of the eights exchange function occurrences in the MRSL specifications. However, we will not elaborate on any details of the simulation run. The output is merely intended to illustrate the sequence of events that take place during a simulation and the large number of variables which play a role in going from one process step to the next even for these simple specifications.

5.3.5 Implementation of Various MRSL Models

Sections 5.1 and 5.2 have shown by rules and examples the major steps (exclusive of analysis) which would be performed upon MRSL specifications if an actual implementation of the AIP model had been produced. We can now make more concrete some of the observations concerning the other three MRSL models already made in section 5.0 and show how the implementation of these other models would compare with that of the AIP model.

As we noted before, certain information concerning messages, R-Nets, and their interrelationships must always be encoded by the final REVS-interpretable RSL-plus-Pascal code into which the MRSL specifications are eventually translated, no matter which of the four models is chosen. We have no choice then but to insert constants into the combined RSL-plus-Pascal code, that is, if we wish also to have a standardized simulation package so that the designer does not have to worry about low level details of the simulation. There are basically three ways to perform this parameterization: (1) to encode information as constant parameters to routines which can interpret them, (2) to initialize reserved variables and arrays with constant values (via assignment statements, e.g. in the SSSTARTUP routine) so that the standardized simulation routines can access the information as necessary during the course of a

simulation, or (3) to pass constants as part of MRSL messages so that the information can be interpreted and routed by the standardized drivers receiving the messages. We might, of course, also want to use a combination of the above three techniques.

The four models can now be viewed in terms of their implementations rather than the source code forms, as in section 5.0. The three techniques above characterize the AIP model (constant parameters to simulation routines), the table model (constants assigned to reserved variables and arrays), and the message model (constants in MRSL messages), respectively. The fourth model, the augmented RSL model, could use either technique (1) or (2) above with the difference that the constants would be automatically supplied during the translation of source MRSL to RSL-plus-Pascal.

In the table model and message model the designer must keep track of these constants by himself. The only real difference, however, between these two and the augmented RSL model is that the designer has at his disposal reserved words and language constructs which are little more than disguised versions of the constants he would otherwise be dealing with in the table model or message model. In any of these three models our standardized simulation procedures, helpful as they may be, relieve only a fraction of the burden from the designer in his tedious remolding of AIP concepts by means of a series of relatively primitive structures and

encodings. Only the AIP model, which would use a source language far removed from RSL/REVS and uncontaminated by its conventions, can serve as part of a reasonable design environment. This conclusion was drawn previously, of course, in section 5.0.6.

5.4 Conclusions

We will not list again the merits of using a design language allied to AIP concepts with MRSL playing an intermediary role. The advantages of this approach and shortcomings of other MRSL models have been adequately summarized in sections 5.0.6 and the immediately preceding 5.3.5. However, we can discuss implementation of the AIP model from a broader perspective than in most of the preceding text of section 5. As we noted much earlier in sections 1 and 2 we did not undertake the development of analysis tools for MRSL to insure the formal properties of sections 3 and 4. Certainly if we adopted the AIP model as we recommended then we would not want to develop such tools because we could analyze the original source language for semantic errors much more easily. We could then translate source code obeying the formal properties into MRSL form and thus guarantee that adherence to the formal properties would be carried over automatically. Indeed, the major steps necessary for performing this translation were outlined in algorithmic form in section 5.1.

If we were instead to use the table model, message model, or augmented RSL model for writing AIP specifications then not only would we impose great burdens on the designer, as we have pointed out repeatedly, but also we would find the job of analyzing the source code much more difficult.

The analysis would be equivalent to an attempt to abstract non-existant AIP specifications from the low-level MRSL primitives and then to analyze the abstracted specifications for validity with respect to the criteria of consistency, completeness, and so on. This is the strongest argument that can be mode to show that development of analysis tools for MRSL itself, in any model, would be a futile exercise.

There is at least one AIP specification language, section 3.4.3.2 of [Fi322], in which the formal properties are quite accessible to verification. The checks which must be performed on the source code written in the syntax given in [Fi322] were listed informally along with the language. Such a list of semantic checks for MRSL (in any model) would be vastly more complex. This observation only serves to reinforce the claim that MRSL is not a suitable medium for the designer to write AIP specifications, although it is suitable as an intermediate language.

6. DDP EXPERIMENT (RR-4)

6.1 Introduction

The ongoing CS-1 DDP experiment by ABMDSC-ATC was selected as the vehicle for this research task. By using the preliminary results of that work, we were able to produce a prototype Data Processing Requirements (DPR) specification in the form of asynchronously interacting processes (AIP) suitable for translation to MRSL for RSL and REVS.

The AIP form of DPR was then partially translated to R-Net form to demonstrate both the MRSL suitability and the relationship between R-Nets and AIP. Each AIP translates into several R-Nets. An MRSL type interpretation for such R-Nets was also developed.

The partial translation of a AIP DPR into a form suitable for the equi-phase simulation discussed in section 3 was carried out by hand, since the required translator has not been implemented. The results of this translation and simulation are described below. They demonstrate the basic simplicity of distributed simulation of the AIP form of specification.

6.2 CS-1 DPR Development

The CS-1 experiment is the first in a series to study specification methodologies in the context of realistic but simplified portions of ABMD constructs. The results should form an unclassified test bed for subsequent research in development methodology.

The CS-1 system is a representative underlay (terminal) defense system for BMD which is capable of detecting targets within an assigned search region, rejecting from consideration objects which are obviously non-threatening, and transferring potentially threatening objects to a yet unspecified tracking system.

The radar used by the CS-1 designers was previously designed and a simulator for it already exists. The experimenters plan to use the existing SETS capability at the BMD-ATC Advanced Research Center for both radar and threat environment modeling.

The unclassified documents referenced in Appendix B were studied in order to understand the engineering design decisions made by the experts in that field. Due to the preliminary nature of these documents and the experimental character of the specifications, numerous inconsistencies, ambiguities, and omissions were discovered. We do not have specific engineering or design experience in the field of radar data processing to enable us to second guess on such specification problems. Consequently, where we were forced to make decisions, we attempted to find the simplest common denominator among the conflicting documents.

One of the advantages of our more formal approach to specifications soon became obvious. Inconsistent or missing information could not be overlooked. At the same time, we were able to develop comparable specifications with essentially the

same information contained in the existing documents. Our first goal was not to extend the design, but was to specify the design at the same level of informational detail in order to demonstrate that formality of expression does not imply an increase in the informational detail.

The importance of the exchange graph (as shown in Appendix B) became obvious in the early stages. The information encoded in such a graph is precisely that which is required to factor the development of the interacting processes without forcing the specification of as yet unknown details. The consistency of the exchange graph with the subsequent design specification can be readily and automatically shown by deriving it from the specification. Although the exchange graph is a formal abstraction of the specification, it plays an important role in the decomposition of the complexity of the development process for the specification. It thus precedes the system specification in our development process. The exchange graph is not itself a system specification since it only defines some of a systems properties.

The specification takes the form of a set of definitions for the processes, sets, and functions required to formally specify the CS-1 system in terms of AIP. These are defined in terms of a set of primitive functions and sets. We deliberately chose to leave these primitives at as high a level as possible, while still allowing the formal process decomposition. The elaboration of

these primitives is left to subsequent phases of development. This choice is consistent with the purpose of requirement specifications as well as illustrating a very high level formal (but not detailed) system specification.

The automated analysis tools for the formal specification properties are still being developed and are not available for current use. Consequently, the AIP type DPR in Appendix B may still have some errors in it. Their complete elimination will await the analysis tool implementations. Note that it is well within the current state of the art to ensure that such a specification does completely and consistently specify the CS-1 system to the level of detail of the primitive sets and functions. This ability alone can prevent many kinds of errors at an early stage of development.

The performance requirements for CS-1 are treated informally and could be incorporated unchanged from the source documents. An approach to formalizing performance requirements was discussed in an earlier section of this report. Those results have not been used, as yet, to improve on our informal approach.

The elaboration of the primitives and node allocation based on performance requirements will lead to a data processing architecture requirements (DPAR) specification that can be automatically shown to be consistent with the DPR and subjected to the same kinds of analysis. This ability automatically eliminates many kinds of errors, and guarantees that decisions in the DPR are preserved in the DPAR. Of course, the DPAR work

may lead to changes in the DPR, but they can at least be kept up to date and consistent.

The AIP type DPR in the appendix is well suited for formal property analysis. Indeed, it was designed to be so. It is also a suitable starting point for the automatic generation of the MRSL equivalent as described in a previous section of this report. That translation, however, is too elaborate to carry out correctly by hand, and the translator has not as yet been implemented. Instead, we will present some manually generated R-Net control graphs for the equivalent forms into the next section. Similarly, we will manually generate an equivalent multi-tasking program (suitable for equi-phase simulation) for a simple example system. The results of that simulation are in Appendix C of this report.

6.3 Analysis

Our analysis of the CS-1 DPR is restricted by the lack of the analysis tool implementations based on the DPR form. We have postponed such implementations until there is agreement on the form of the specifications. We have manually carried out two types of analysis. Although human fallibility may have introduced or left undetected errors, the results are illustrative and useful.

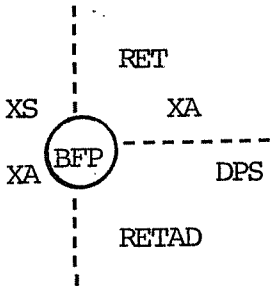
6.3.1 AIP Relationship to RSL

The connection between the AIP model and the RSL model may be clarified by using a part of the CS-1 DPR as an illustration. We will show the results of the transformation of AIP to interface nets (similar to R-Nets) for the processes BFP and ROG.

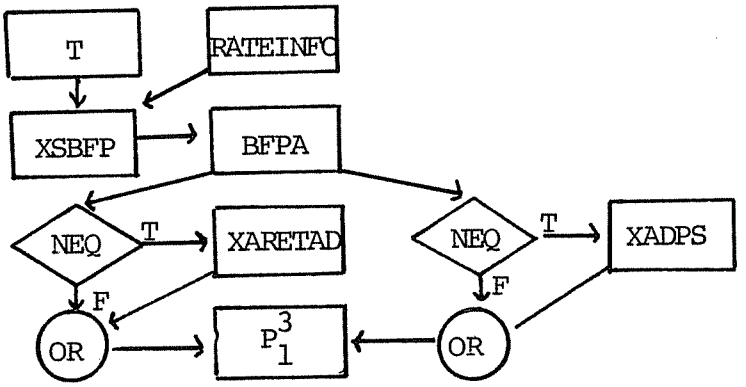
First we must give an interpretation to the AIP. For this illustration we will assume that evaluation time for a process is an attribute of a realization of that process. Thus the AIP is itself timeless (a non-deterministic (ND) interpretation) and all possible interactions will occur at some relative process rates. (The inclusion of time as a formal attribute of AIP is being carried out in task RR-1.) Subject to this interpreation, the following shows equivalence transformations of the AIP into a purely mathematical function (to be evaluated in the ND interpreter) and into interface nets that correspond to the R-Nets of REVS (to be evaluated in the REVS simulator).

6.3.1.1 BFP Process Transformations

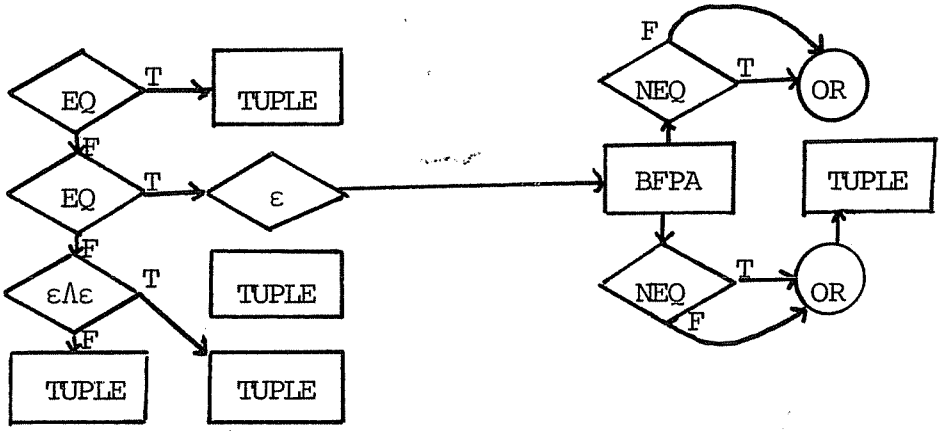
The BFP process of the CS-1 specification can be described as an exchange graph, as a control graph, as an equivalent algebraic function control graph, or as an equivalent interface net similar to R-Net structures. All of these forms are shown in Figure 6.1. The corresponding specifications in mathematical form are given below.



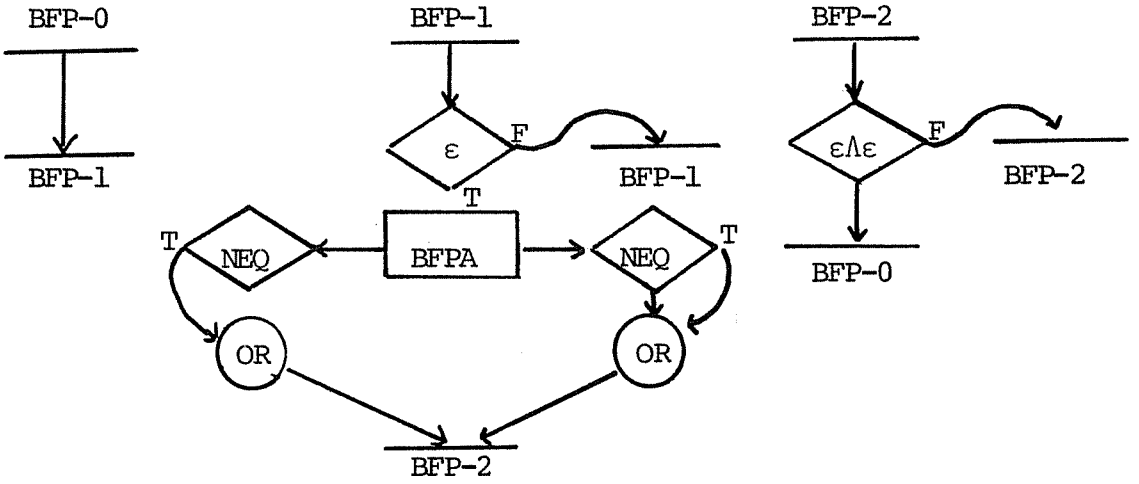
a) Exchange Graph



b) Control Graph for Process



c) Control Graph for Function



d) Control Graph for Interface Net

Figure 6-1: ND Interpretation of BFP Process

Process Form:

The process form is given in the CS-1 DPR.

Non-Deterministic Function Form:

Set: Σ'_{BFP}: ({0},RATEINFO)

$\cup$({1},RATEINFO,{('XS','BFP',T)} $\cup$ RET)

$\cup$({2},RATEINFO,{('XA',RETAD',T)} $\cup$ RETAD,

{('XA','DPS',T)} $\cup$ DPS)

Function: $F'_{BFP} : \Sigma'_{BFP} \rightarrow \Sigma'_{BFP}$

$[\sigma_1=0:(1,\sigma_2,('XS','BFP',T)),$

$\sigma_1=0:[\sigma_3 \in RET:UBFPA(\sigma_2,\sigma_3)),(1,\sigma_2,\sigma_3)]$

$[\sigma_3 \in RETAD \wedge \sigma_4 \in DPS:(0,\sigma_2),(2,\sigma_2,\sigma_3,\sigma_4)]]$

UBFPA:(RATEINFO,RETAD,DPS) $\rightarrow$ ({2},RATEINFO,{('XA','RETAD',T)},

{('XA','DPS',T)})

$(2,\sigma_1,[\sigma_2 \neq T:('XA','RETAD',\sigma_2),],[\sigma_3 \neq T:('XA','RETAD',\sigma_3),])$

Interface Net Form:

Net : BFP_0

Interface : RATEINFO

Outerfaces: BFP_1

Procedure : OUTERFACE_BFP_1:=(INTERFACE_BFP_0,('XS','BFP',T))

Net : BFP_1

Interface : (RATEINFO,{('XA','BFP',T)} ∪ RET)

Outerfaces: BFP_2

Procedure : (σ_1,σ_2):=INTERFACE_BPF_1

$[\sigma_2 \epsilon$ RET:$(\sigma_1,\sigma_3,\sigma_4)$:=BFPA$(\sigma_1,\sigma_2)$;

OUTERFACE_BFP_2:=$(\sigma_1,[\sigma_3 \neq$T:('XA','RETAD',σ_3),],

$[\sigma_4 \neq$ T:('XA','DPS',σ_4),]),(σ_1,σ_2)]

Net : BFP_2

Interface : (RATEINFO,{('XA','RETAD',T)} ∪ RETAD,

{('XA','DPS',DPS)} ∪DPS)

Outerfaces: DFP_2,DFP_2

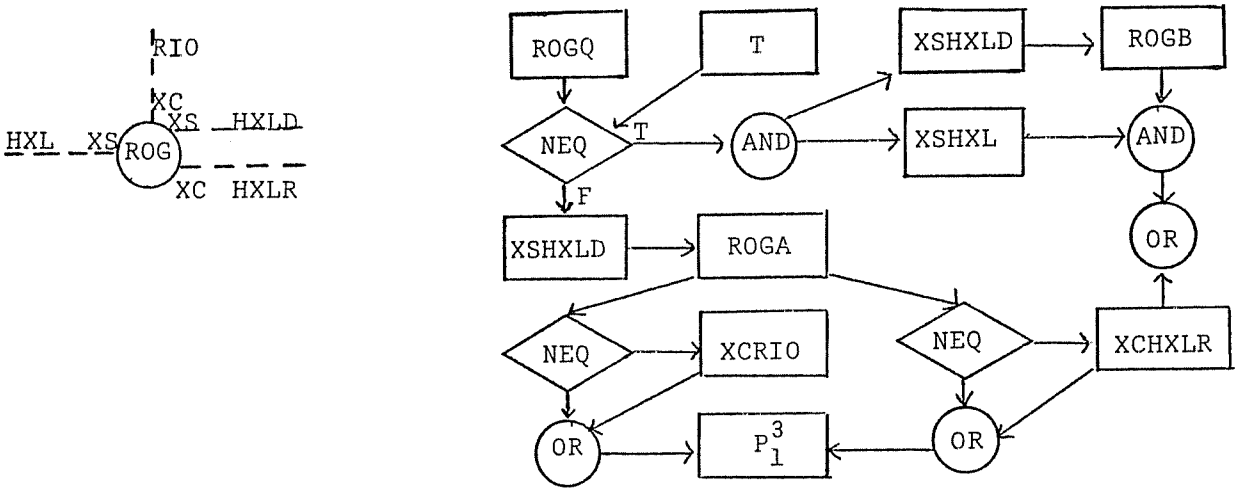
Procedure : $(\sigma_1,\sigma_2,\sigma_3)$:=INTERFACE_BFP_2

$[\sigma_2 \epsilon$ RETAD $\wedge$ $\sigma_3 \epsilon$ DPS:OUTERFACE_BFP_0:=σ_1,

OUTERFACE_BFP_2:=$(\sigma_1,\sigma_2,\sigma_3)$]

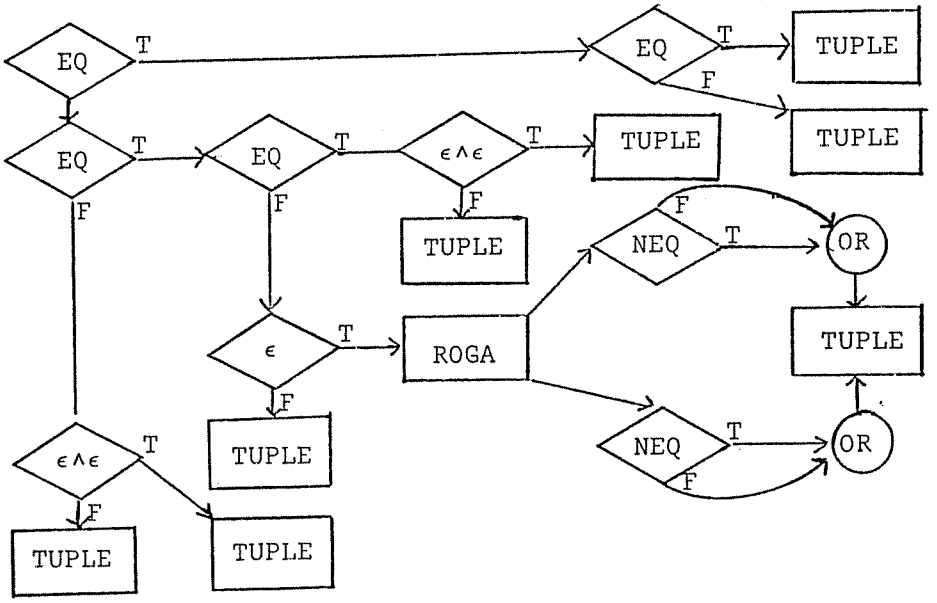
6.3.1.2 ROG Process Transformations

We can similarly transform the ROG process and obtain the following results, as shown in Figure 6-2.



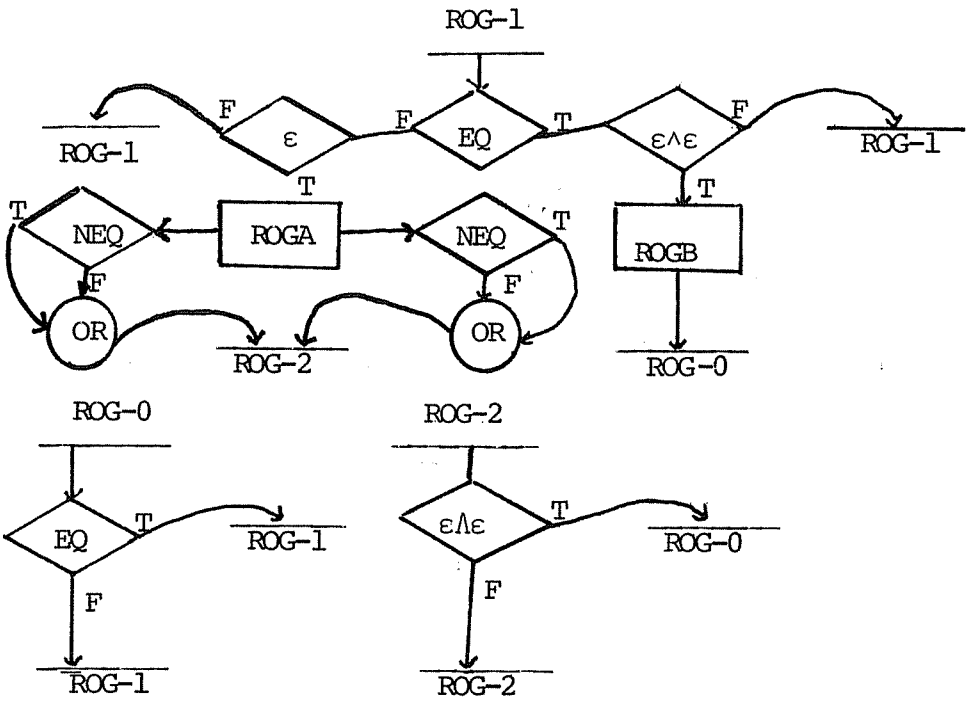
a) Exchange Graph

b) Control Graph for Process



c) Control Graph for Function

Fig. 6-2: ND Interpretation of ROG Process



d) Control Graph for Interface Nets

Process Form

The process form is given in the CS-1 DPR.

Non-Deterministic Function Form:

Set: Σ'_{ROG}: ({0},ROGW,HXLQ)

$\cup$ ({1},{1},('XS','HXL',T)$\cup$HXL,HXLQ,('XS','HXLD',T)$\cup$HXLD)

$\cup$ ({1},{2},ROGQ,HXLQ,('XS','HXLD',T)$\cup$HXLD)

$\cup$ ({2},(ROGQ,HXLQ),('XC','RIO',RIO)$\cup$BOOLEAN,

('XC','HXLR',HXLR)$\cup$BOOLEAN)

Function: $F'_{ROG}: \Sigma'_{ROG} \rightarrow \Sigma'_{ROG}$

$[\sigma_1=0:[\sigma_2=T:(1,1('XS','HXL',T),HXLQ,('XS','HXLD',T)),$

$(1,2,ROGQ,HXLQ,('XS','HXLD',T))],$

$\sigma_1=1:[\sigma_2=1:[\sigma_3 \epsilon HXL \wedge \sigma_5 \epsilon HXLD:(0,\sigma_3,ROGB(\sigma_4,\sigma_5)),(1,1,\sigma_3,\sigma_4,\sigma_5)],$

$[\sigma_5 \epsilon HXLD: ROGA(ROGA(\sigma_3,\sigma_4,\sigma_5),(1,2,\sigma_3,\sigma_4,\sigma_5)]],$

$[\sigma_3 \epsilon BOOLEAN \wedge \sigma_4 \epsilon BOOLEAN:\sigma_2,(2,\sigma_2,\sigma_3,\sigma_4)]]$

$\cup ROGA:(ROGQ,HXLQ,RIO,HXLR) \rightarrow (ROGQ,HXLQ)$

$(2,(ROGQ,HXLQ),[RIO \neq T:('XC','RIO',RIO),],$

$[HXLR \neq T:('XC','HXLR',HXLR),])$

Interface Net Form:

Net : ROG_0

Interface : (ROGQ,HXLQ)

Outerfaces: ROG_1

Procedure :

(σ_1,σ_2):=INTERFACE_ROG_0

OUTERFACE_ROG_1:=$[\sigma_1=T:(1,('XS','HXL',T),\sigma_2,('XS','HXLD',T)),$

$(2,\sigma_1,\sigma_2,('XS','HXLD',T))]$

Net : ROG_1

Interface : ({1},('XS','HXL',T) ∪ HXL,HXLQ,('XS','HXLD',T) ∪ HXLD)

∪ ({2},ROGQ,HXLQ,('XS',HXLD',T) ∪ HXLD)

Outerfaces: ROG_0,ROG_1,ROG_2

Procedure :

$(\sigma_1,\sigma_2,\sigma_3,\sigma_4)$:=INTERFACE_ROG_1

$[\sigma_1$=1:$[\sigma_2 \in$ HXL $\wedge\ \sigma_4 \in$ HXLD:OUTERFACE_ROG_0:=$(\sigma_2$,ROGB$(\sigma_3,\sigma_4))$,

OUTERFACE_ROG_1:=$(1,\sigma_2,\sigma_3,\sigma_4)]$,

$[\sigma_4 \in$ HXLD:$(\sigma_1,\sigma_2,\sigma_3,\sigma_4)$:=ROGA$(\sigma_2,\sigma_3,\sigma_4)$;

OUTERFACE_ROG_2:=$((\sigma_1,\sigma_2),[\sigma_3 \neq$T:('XC','RIO',$\sigma_3$),],

$[\sigma_4 \neq$T:('XC','HXLR',σ_4),],

OUTERFACE_ROG_1:=$(2,\sigma_2,\sigma_3,\sigma_4)]]$

Net : ROG_2

Interface:((ROGQ,HXLQ),('XC','RIO',RIO) ∪ BOOLEAN,

('XC','HXLR',HXLR) ∪ BOOLEAN)

Outerfaces: ROG_0,ROG_2

Procedure :

$[\sigma_2 \in$ BOOLEAN $\wedge\ \sigma_3 \in$ BOOLEAN:OUTERFACE_ROG_0:=σ_1,

OUTERFACE_ROG_2:=$(\sigma_1,\sigma_2,\sigma_3)]$

6.3.2 Multi-Tasking Analysis

The CS-1 DPR transformations to RSL forms are well suited to conventional (sequential) discrete event simulation. Each event is linearly ordered in simulated (as well as in simulation) time. If we want to carry out our simulation of the intrinsically distributed CS-1 system using a distributed simulator, the linear form is a very bad starting point since we can evaluate only one event at a time.

The AIP specifications constrain only required precedences without forcing an arbitrary evaluation control sequence on the events. We can transform the AIP into multi-tasking programs suitable for evaluation on distributed systems and exploit the potential parallelism of the simulated system. Each possible parallel evaluation forms a task that is scheduled for independent execution when its precedences are satisfied.

We have developed a "quick and dirty" single processor simulator of such a multi-tasking program on the PDP 11/45 under the UNIX operating system. This implementation is only intended to support experiments in distributed simulation concepts, and will have only a transient existence.

The simulator is based on a doubly linked list structure in a shared memory, multi-processor machine (that currently has only one processor). The AIP are translated to a sequence of program statements in the C programming language. These statements are then combined with the simulator program (also written in C) and compiled to form the simulation program.

The interpretation of exchange functions is done in an equi-phase mode. All processes that can run, will run to their next interaction prior to any interactions taking place. Thus the effect of an infinite number of processors is simulated, without introducing time of events.

The execution of an exchange function simply makes an initiation entry in a list and its containing task goes to sleep until the interaction is completed in a subsequent phase of evaluation.

The translation is unnecessarily inefficient and most of the parallelism is not, in fact, used by the current simulator program. The results, however, are independent of the simulator parallelism or lack thereof.

The details of the multi-tasking form are not important, but a flavor of them can be obtained from the following, informal, description of the translated codes.

The simulator program provides a small set of service functions used by the multi-tasking statements to manipulate task structures. The purpose of each is briefly described below.

newtask will create a new task list with i current entry index and actual parameter list j.

par will return with the index to the i actual parameter for the current task.

res will return with the index to the i result list cell.

awake puts task i at end of PEND list

sinit will scan its argument string.

The character array x will be decoded to control the creation of a list from the standard input.

retinit initializes for birth of sub-tasks by creating a result list of i NL cells.

pinit initializes i new processes as sub-tasks to the main routine. j is index to make entry points unique.

x is a 'formatting' string.

tiinit initializes i sub-tasks for a tuple.

j is an index to make entry points unique.

supdt moves step results state to initial state of the process.

stuff returns to sub-task parent with results.

done returns a function value to invoker.

dot will set current task entry point to m and initialize a new task.

rval returns the actual parameter i as current task results.

The translated statements are described as the results of applying a "value" function to the corresponding AIP form, as follows:

value (specification)
n is the number of processes. en is the defining expression for the corresponding process.
x is a string that specifies the creation of the initial states of the processes.
The specification is evaluated by creating and executing a task tree for each process.

```
tel:  ;
      pinit(n,x);
      goto sched;
te(2+1):  value(el);
          if (t[p[task]]!=NL){supdt( );go to te(2+1);}
          stuff(1); go to te2;
     ...
te(2+n):  value(en);
          if (t[p[task]]!=NL){supdt( ); go to te(2+n);}
          stuff(n);
te2:  if (q[task ]]go to sched;
      value (tuple)
     If the tuple is of size 0, nothing is generated.
     If the tuple is of size 1, the value ((el)) is value (el);
     If the tuple is larger than 1, we generate multiple sub-tasks
to evaluate components in parallel.
     n is the size of the tuple.  m  is an index to make task entry
points unique.  tem is a unique label.
tinit (n,m);
go to sched;
te(m+1):  value (el);  stuff(1); go to tem;
     ...
te(m+n):  value(en);  stuff(n);
tem:  if (q[task]) go to sched;
      value (selector)
      n is the number of expressions in the selector.
      p[i] are the predicates.  e[i] are the associated expressions,
tem is a unique label.
      value(p1);
      if (d[p[task]]){value(el); go to tem;}
     ...
     value (en);
tem:;
     value (parinvoc)
     i indexes the i-th parameter.
     rval (i);
     value (function)
     f is the name of the function and e is the definition expression
for f.  If f is a primitive function, the defining expression will be
supplied by translator.
f:  value (e);
    done( );
    go to retsw;
    value (funct-invoc)
    tuple is the actual parameter list for this invocation of function
f.  n  is an index used to generate a unique label for retsw.
```

```
     value (tuple);
     dot (m);
     go to f;
tem: ;
```

If the function invoked is a 'built in' one, the code will be

```
     value (tuple);
     f( );
```

If the function invoked is an exchange function, instead of transferring to the function name as entry point, the registers are located with type t, and class c, values and control is transferred to exchange as

```
     value (e);
     dot(m);
     val=t;  tmp=c;  go to exchange;
tem: ;
```

Because of the hand labor involved (and the resulting invalidating uncertainties) in the manual translation to the multi-tasking form, we developed, translated, and simulated a simpler, but non-trivial, example system. The results of that work are given as Appendix C to this report, and serve to illustrate the concepts of the equi-phase (timeless and distributed) simulation/emulation developed in section 3.

6.4 Conclusions

The major conclusion from this experiment is that AIP specifications can be used at the DPR and DPAR level without extraneous details. The equivalent RSL or multi-tasking forms can be automatically produced by machine and are far more suited for machine usage than they are for interfacing with a designer.

A corollary to the above conclusion is that the RSL/REVS system can be used for distributed systems with few changes. The major changes is only the addition to an AIP translator to R-Net form. This translator is required because the direct generation of the R-Nets for asynchronously interacting processes is much too complex for a designer to carry out reliably. We need the AIP form also for efficient analysis. We can let the computer generate the R-Nets and obey all the rules to keep them asynchronously consistent. The additional analysis tools required can be included using the data base extension mechanisms already supported by RSL/REVS.

The equi-phase simulator/emulator can be used to analyze specifications, even at the DPR level, and provide valuable feedback to the designers. Its use (with the translator above) is simple, quick, and automatic since it operates only with the specification itself and the designer's selection of intializations.

The use of the translator and the equi-phase simulator/emulator now gives the designer the same ability to check the specifications syntactically and semantically that programmers have had using compilers. This ability is even more important to designers than it is to programmers.

The final conclusion from this experiment is that it is far from complete. We have only begun to exploit its possibilities as a research and demonstration vehicle. It has already been a valuable experience.

Appendix A--MRSL Examples

All of the listings reproduced in this appendix were taken from a single continuous listing made by a printer which unfortunately changed certain special characters in the character set from their original representations. The following correspondences should be observed:

character	intended representation
'	[
|	]
#	'
¢	<

A.1 Sample MRSL Code

```
XX 000  REVS   BASELINE VERSION = 12, (DATE=02/02/78,  TIME=18.03.35)
        RSL.

XX 001  FUNCTION RSL      INITIATED.  **********************************
INPUT_INTERFACE:
          IAB.
     CONNECTS TO: SUBSYSTEM:
          SAB.
    PASSES: MESSAGE:
          AB.
OUTPUT_INTERFACE:
          IAE1_2.
     CONNECTS TO: SUBSYSTEM:
          SAE1_2.
    PASSES: MESSAGE:
          AE1_2.
OUTPUT_INTERFACE:
          IAE3_4.
     CONNECTS TO: SUBSYSTEM:
               SAE3_4.
    PASSES: MESSAGE:
          AE3_4.
INPUT_INTERFACE:
          IBB.
     CONNECTS TO: SUBSYSTEM:
          SBB.
    PASSES: MESSAGE:
          BB.
OUTPUT_INTERFACE:
          IBE.
     CONNECTS TO: SUBSYSTEM:
          SBE.
    PASSES: MESSAGE:
          BE.
INPUT_INTERFACE:
          ICB.
     CONNECTS TO: SUBSYSTEM:
          SCB.
    PASSES: MESSAGE:
          CB.
OUTPUT_INTERFACE:
          ICE.
     CONNECTS TO: SUBSYSTEM:
          SCE.
    PASSES: MESSAGE:
          CE.
INPUT_INTERFACE:
          IDB.
     CONNECTS TO: SUBSYSTEM:
          SDB.
    PASSES: MESSAGE:
          DB.
OUTPUT_INTERFACE:
          IDE1_2.
     CONNECTS TO: SUBSYSTEM:
          SDE1_2.
    PASSES: MESSAGE:
          DE1_2.
```

```
INPUT_INTERFACE:
        IEB.
    CONNECTS TO: SUBSYSTEM:
        SEB.
    PASSES: MESSAGE:
        EB.
OUTPUT_INTERFACE:
        IEE.
    CONNECTS TO: SUBSYSTEM:
        SEE.
    PASSES: MESSAGE:
        EE.
R_NET: A.
    ENABLED BY: INPUT_INTERFACE:
        IAB.
    STRUCTURE:
        INPUT_INTERFACE: IAB
        DO
            DO
                ALPHA: ALA1
            AND
                ALPHA: ALA2
            END
            ALPHA: ALA1_2
                OUTPUT_INTERFACE: IAE1_2
        AND
            DO
                ALPHA: ALA3
            AND
                ALPHA: ALA4
            END
            ALPHA: ALA3_4
            OUTPUT_INTERFACE: IAE3_4
        END
    END.
R_NET: B.
    ENABLED BY: INPUT_INTERFACE:
        IBB.
        STRUCTURE:
        INPUT_INTERFACE: IBB
            DO
                ALPHA: ALB1
            AND
                ALPHA: ALB2
            END
            ALPHA: ALB3
            ALPHA: ALBT
            OUTPUT_INTERFACE: IBE
    END.
R_NET: C.
    ENABLED BY: INPUT_INTERFACE:
        ICB.
    STRUCTURE:
        INPUT_INTERFACE: ICB
        DO
    IF (CONDAE2 = 1)
                ALPHA: ALC1
    OR (CONDAE2 = 2)
```

```
                    ALPHA: ALC2
        OTHERWISE
                    ALPHA: ALC3
                END
            AND
                ALPHA: ALC4
            END
            ALPHA: ALC5
            ALPHA: ALCT
            OUTPUT_INTERFACE: ICE
        END.
    R_NET: D.
        ENABLED BY: INPUT_INTERFACE:
            IDB.
        STRUCTURE:
            INPUT_INTERFACE: IDB
            DO
                ALPHA: ALD1
                AND
                    ALPHA: ALD2
                END
                ALPHA: ALDT
                OUTPUT_INTERFACE: IDE1_2
        END.
    R_NET: E.
        ENABLED BY: INPUT_INTERFACE:
            IEB.
        STRUCTURE:
            INPUT_INTERFACE: IEB
    ALPHA: ALE1
            ALPHA: ALET
            OUTPUT_INTERFACE: IEE
        END.
    MESSAGE: AE1_2.
        MADE BY:
        DATA: TIMEAE1
        DATA: TIMEAE2
        DATA: COND2
        DATA: LVA2.
    MESSAGE: AE3_4.
        MADE BY:
        DATA: TIMEAE3
        DATA: TIMEAE4
        DATA: EXE3
        DATA: EXE4.
    MESSAGE: AB.
        MADE BY:
        DATA: TIMEAB.
    MESSAGE: BB.
        MADE BY:
        DATA: TIMEBB1
        DATA: TIMEBB2
        DATA: EXA1
        DATA: EXA2.
    MESSAGE: BE.
        MADE BY:
        DATA: TIMEBE
        DATA: LVA7.
```

```
MESSAGE: CB.
      MADE BY:
      DATA: TIMECB1
      DATA: TIMECB2
      DATA: CONDAE2
      DATA: EXA3
      DATA: TIMECB3
      DATA: EXA4.
MESSAGE: CE.
      MADE BY:
      DATA: TIMECE.
MESSAGE: DB.
      MADE BY:
      DATA: TIMEDB1
      DATA: TIMEDB2
      DATA: EXB1
      DATA: EXB2.
MESSAGE: DE1_2.
      MADE BY:
      DATA: TIMEDE1
      DATA: TIMEDE2
      DATA: LVB1
      DATA: LVB2.
MESSAGE: EB.
      MADE BY:
      DATA: TIMEEB1
      DATA: TIMEEB2
      DATA: EXB3
      DATA: EXB4.
MESSAGE: EE.
      MADE BY:
      DATA: TIMEEE1
      DATA: TIMEEE2
      DATA: EXE1
      DATA: EXE2.
ALPHA: ALA1.
      GAMMA:
      "BEGIN
      IVA1 := SVA1 + 1;
      ELAPA1 := 3;
      WRITELN (OUTPUT,
      # END ALA1#, IVA1);
      WRITELN (OUTPUT, # ++++++++++++#)
      END;".
      INPUTS:
            DATA: SVA1.
      OUTPUTS:
            DATA: IVA1
            DATA: ELAPA1.
ALPHA: ALA2.
      GAMMA:
      "BEGIN
      LVA2 := SVA2 + 1;
      COND2 := SVA2 MOD 3 + 1;
      ELAPA2 := 2;
      WRITELN (OUTPUT,
      # END ALA2#, LVA2, COND2);
      WRITELN (OUTPUT, # ++++++++++++#)
```

```
        END;".
        INPUTS:
             DATA: SVA2.
        OUTPUTS:
             DATA: LVA2
             DATA: COND2
        DATA: ELAPA2.
   ALPHA: ALA3.
        GAMMA:
        "BEGIN
        IVA3 := SVA3 - SVA2 + 100;
        ELAPA3 := 1;
        WRITELN (OUTPUT,
        # END ALA3#, IVA3);
        WRITELN (OUTPUT, # ++++++++++++#)
        END;".
        INPUTS:
             DATA: SVA3
        DATA: SVA2.
        OUTPUTS:
             DATA: IVA3
             DATA: ELAPA3.
   ALPHA: ALA4.
        GAMMA:
        "BEGIN
        SVAA1 := SVA4 - SVA3 + 1;
        IVA4 := SVA4 - SVA3 + 200;
        ELAPA4 := 1;
        WRITELN (OUTPUT,
        # END ALA4#, SVAA1, IVA4);
        WRITELN (OUTPUT, # ++++++++++++#)
        END;".
        INPUTS:
             DATA: SVA3
             DATA: SVA4.
        OUTPUTS:
        DATA: SVAA1
             DATA: IVA4
             DATA: ELAPA4.
   ALPHA: ALA1_2.
        GAMMA:
        "BEGIN
        TIMEAE1 := TIMEAB + ELAPA1;
        TIMEAE2 := TIMEAB + ELAPA2;
        WRITELN (OUTPUT,
        # END ALA1_2#, TIMEAE1, TIMEAE2);
        WRITELN (OUTPUT, # ++++++++++++#)
        END;".
        INPUTS:
12           DATA: TIMEAB
11           DATA: ELAPA1
10           DATA: ELAPA2.
 9      OUTPUTS:
 8           DATA: TIMEAE1
 7           DATA: TIMEAE2.
 6      FORMS: MESSAGE: AE1_2.
 5 ALPHA: ALA3_4.
 4      GAMMA:
 3
```

```
        "BEGIN
        TIMEAE3 := TIMEAB + ELAPA3;
        TIMEAE4 := TIMEAB + ELAPA4;
        EXE3 := IVA3;
        EXE4 := IVA4;
        WRITELN(OUTPUT,
        # END ALA3_4#, TIMEAE3, TIMEAE4);
        WRITELN (OUTPUT, # ++++++++++++#)
        END;".
        INPUTS:
             DATA: TIMEAB
             DATA: ELAPA3
        DATA: ELAPA4.
        OUTPUTS:
        DATA: EXE3
        DATA: EXE4
             DATA: TIMEAE3
             DATA: TIMEAE4.
        FORMS: MESSAGE: AE3_4.
   ALPHA: ALB1.
        GAMMA:
        "BEGIN
        LVA6 := EXA1;
        ELAPB1 := 3;
        WRITELN (OUTPUT,
        # END ALB1#, LVA6);
        WRITELN (OUTPUT, # ++++++++++++#)
        END;".
        INPUTS:
             DATA: EXA1.
        OUTPUTS:
             DATA: LVA6
             DATA: ELAPB1.
   ALPHA: ALB2.
        GAMMA:
        "BEGIN
        LVA7 := EXA2;
        ELAPB2 := 4;
        WRITELN (OUTPUT,
        # END ALB2#, LVA7);
        WRITELN (OUTPUT, # ++++++++++++#)
        END;".
        INPUTS:
             DATA: EXA2.
        OUTPUTS:
             DATA: LVA7
             DATA: ELAPB2.
   ALPHA: ALB3.
        GAMMA:
12      "BEGIN
11      SVA3 := (LVA6 + LVA7) MOD (IVA3 + 1);
10      ELAPB3 := 1;
9       WRITELN (OUTPUT,
8       #END ALB3#, SVA3);
7       WRITELN (OUTPUT, # ++++++++++++#)
6       END;".
5       INPUTS:
4            DATA: LVA6
3
```

```
            DATA: LVA7
            DATA: IVA3.
      OUTPUTS:
            DATA: SVA3
            DATA: ELAPB3.
ALPHA: ALBT.
      GAMMA:
      "BEGIN
      TIMEALB1 := TIMEBB1 + ELAPB1;
      TIMEALB2 := TIMEBB2 + ELAPB2;
      IF TIMEALB1 > TIMEALB2 THEN
            TIMEBE := TIMEALB1 + ELAPB3
      ELSE TIMEBE := TIMEALB1 + ELAPB3;
      WRITELN (OUTPUT,
      # END ALBT#, TIMEBE);
      WRITELN (OUTPUT, # ++++++++++++#)
      END;".
      INPUTS:
            DATA: TIMEBB1
            DATA: ELAPB1
            DATA: TIMEBB2
            DATA: ELAPB2
            DATA: ELAPB3.
      OUTPUTS:
      DATA: TIMEALB1
      DATA: TIMEALB2
            DATA: TIMEBE.
      FORMS: MESSAGE: BE.
ALPHA: ALC1.
      GAMMA:
      "BEGIN
      SVA2 := IVA2 + EXA3;
      LVA5 := SVA2;
      ELAPC1_2_3 := 5;
      WRITELN (OUTPUT,
      # END ALC1#, SVA2);
      WRITELN (OUTPUT, # ++++++++++++#)
      END;".
      INPUTS:
            DATA: IVA2
            DATA: EXA3.
      OUTPUTS:
      DATA: LVA5
            DATA: SVA2
            DATA: ELAPC1_2_3.
ALPHA: ALC2.
      GAMMA:
      "BEGIN
      SVA2 := (IVA2 + EXA3) MOD (SVA3 + 1);
      LVA5 := SVA2;
      ELAPC1_2_3 := 6;
      WRITELN (OUTPUT,
      # END ALC2#, SVA2);
      WRITELN (OUTPUT, # ++++++++++++#)
      END;".
      INPUTS:
           DATA: IVA2
           DATA: EXA3
```

```
            DATA: SVA3.
      OUTPUTS:
      DATA: LVA5
            DATA: SVA2
            DATA: ELAPC1_2_3.
ALPHA: ALC3.
      GAMMA:
      "BEGIN
      SVA2 := (IVA2 + EXA3) MOD (SVA4 + 1);
      LVA5 := SVA2;
      ELAPC1_2_3 := 7;
      WRITELN (OUTPUT,
      # END ALC3#, SVA2);
      WRITELN (OUTPUT, # ++++++++++++#)
      END;".
      INPUTS:
            DATA: IVA2
            DATA: EXA3
            DATA: SVA4.
      OUTPUTS:
      DATA: LVA5
            DATA: SVA2
            DATA: ELAPC1_2_3.
ALPHA: ALC4.
      GAMMA:
      "BEGIN
      IVA8 := IVA4 MOD (EXA4 + 1);
      ELAPC4 := 2;
      WRITELN (OUTPUT,
      # END ALC4#, IVA8);
      WRITELN (OUTPUT, # ++++++++++++#)
      END;".
      INPUTS:
            DATA: IVA4
            DATA: EXA4.
      OUTPUTS:
            DATA: IVA8
            DATA: ELAPC4.
ALPHA: ALC5.
      GAMMA:
      "BEGIN
      SVA1 := SVAA1;
      SVA4 := (IVA1 + IVA8) MOD (LVA5 + 1);
      ELAPC5 := 1;
      WRITELN (OUTPUT,
      # END ALC5#, SVA4);
      WRITELN (OUTPUT, # ++++++++++++#)
      END;".
      INPUTS:
      DATA: SVAA1
            DATA: IVA8
            DATA: IVA1
            DATA: LVA5.
      OUTPUTS:
            DATA: SVA1
            DATA: SVA4
            DATA: ELAPC5.
ALPHA ALCT.
```

```
        GAMMA:
        "BEGIN
        TIMEC13 := TIMECB2 + ELAPC1_2_3;
        TIMEC4 := TIMECB3 + ELAPC4;
        IF TIMECB1 > TIMEC13 THEN
             TIMECE := TIMECB1
        ELSE TIMECE := TIMEC13;
        IF TIMEC4 > TIMECE THEN
             TIMECE := TIMEC4;
        TIMECE := TIMECE + ELAPC5;
        WRITELN (OUTPUT,
        # END ALCT#, TIMECE);
        WRITELN (OUTPUT, # ++++++++++++#)
        END;".
        INPUTS:
             DATA: TIMECB1
             DATA: TIMECB2
             DATA: TIMECB3
             DATA: ELAPC4
        DATA: ELAPC5
             DATA: ELAPC1_2_3.
        OUTPUTS:
        DATA: TIMEC13
        DATA: TIMEC4
             DATA: TIMECE.
        FORMS: MESSAGE: CE.
ALPHA: ALD1.
        GAMMA:
        "BEGIN
        LVB1 := EXB1 + 1;
        ELAPD1 := 10;
        WRITELN (OUTPUT,
        # END ALD1#, LVB1);
        WRITELN (OUTPUT, # ++++++++++++#)
        END;".
        INPUTS:
             DATA: EXB1.
        OUTPUTS:
             DATA: ELAPD1
             DATA: LVB1.
ALPHA: ALD2.
        GAMMA:
        "BEGIN
        LVB2 := EXB2 + 1;
        ELAPD2 := 12;
        WRITELN (OUTPUT,
        # END ALD2#, LVB2);
        WRITELN (OUTPUT, # ++++++++++++#)
        END;".
        INPUTS:
             DATA: EXB2.
        OUTPUTS:
             DATA: ELAPD2
             DATA: LVB2.
ALPHA: ALDT.
        GAMMA:
        "BEGIN
        TIMEDE1 := TIMEDB1 + ELAPD1;
```

```
        TIMEDE2 := TIMEDB2 + ELAPD2;
        WRITELN (OUTPUT,
        # END ALDT#, TIMEDE1, TIMEDE2);
        WRITELN (OUTPUT, # ++++++++++++#)
        END;".
        INPUTS:
             DATA: TIMEDB1
             DATA: TIMEDB2
             DATA: ELAPD1
             DATA: ELAPD2.
        OUTPUTS:
             DATA: TIMEDE1
             DATA: TIMEDE2.
        FORMS: MESSAGE: DE1_2.
  ALPHA: ALE1.
        GAMMA:
        "BEGIN
        SVB1 := EXB3 + EXB4;
        SVB2 := EXB3 - EXB4;
        EXE1 := SVB1;
        EXE2 := SVB2;
        ELAPE1 := 6;
        WRITELN (OUTPUT,
        # END ALE1#, SVB1, SVB2);
        WRITELN (OUTPUT, # ++++++++++++#)
        END;".
        INPUTS:
             DATA: EXB3
             DATA: EXB4.
        OUTPUTS:
             DATA: SVB1
             DATA: SVB2
             DATA: EXE1
             DATA: EXE2
             DATA: ELAPE1.
  ALPHA: ALET.
        GAMMA:
        "BEGIN
        IF TIMEEB1 > TIMEEB2 THEN
             TIMEEE1 := TIMEEB1 + ELAPE1 ELSE
             TIMEEE1 := TIMEEB2 + ELAPE1;
        TIMEEE2 := TIMEEE1;
        WRITELN (OUTPUT,
        # END ALET#, TIMEEE1);
        WRITELN (OUTPUT, # ++++++++++++#)
        END;".
        INPUTS:
             DATA: TIMEEB1
             DATA: TIMEEB2
             DATA: ELAPE1.
        OUTPUTS:
             DATA: TIMEEE1
             DATA: TIMEEE2.
        FORMS: MESSAGE: EE.
  DATA: SVA1.
        INITIAL_VALUE:
             1000 .
        USE: GAMMA.
```

```
            LOCALITY: GLOBAL.
      TYPE: INTEGER.
      DATA: SVAA1.
      USE: GAMMA.
            LOCALITY: GLOBAL.
      TYPE: INTEGER.
DATA: SVA2.
      INITIAL_VALUE:
            2000 .
      USE: GAMMA.
            LOCALITY: GLOBAL.
      TYPE: INTEGER.
DATA: SVA3.
      INITIAL_VALUE:
            3000 .
      USE: GAMMA.
            LOCALITY: GLOBAL.
      TYPE: INTEGER.
DATA: SVA4.
      INITIAL_VALUE:
            4000 .
      USE: GAMMA.
            LOCALITY: GLOBAL.
      TYPE: INTEGER.
DATA: IVA1.
      USE: GAMMA.
            LOCALITY: GLOBAL.
      TYPE: INTEGER.
DATA: IVA2.
      USE: GAMMA.
            LOCALITY: GLOBAL.
      TYPE: INTEGER.
DATA: IVA3.
      USE: GAMMA.
            LOCALITY: GLOBAL.
      TYPE: INTEGER.
DATA: IVA4.
      USE: GAMMA.
            LOCALITY: GLOBAL.
      TYPE: INTEGER.
DATA: IVA8.
      USE: GAMMA.
            LOCALITY: GLOBAL.
      TYPE: INTEGER.
DATA: SVB1.
      INITIAL_VALUE:
            100000 .
      USE: GAMMA.
            LOCALITY: GLOBAL.
      TYPE: INTEGER.
DATA: SVB2.
      INITIAL_VALUE:
            200000 .
      USE: GAMMA.
            LOCALITY: GLOBAL.
      TYPE: INTEGER.
DATA: LVA2.
      USE: GAMMA.
```

```
        TYPE: INTEGER.
        LOCALITY: LOCAL.
    DATA: LVA5.
        USE: GAMMA.
        LOCALITY: LOCAL.
        TYPE: INTEGER.
    DATA: LVA6.
        USE: GAMMA.
        LOCALITY: LOCAL.
        TYPE: INTEGER.
    DATA: LVA7.
        USE: GAMMA.
        LOCALITY: LOCAL.
        TYPE: INTEGER.
    DATA: COND2.
        USE: GAMMA.
        LOCALITY: LOCAL.
        TYPE: INTEGER.
    DATA: CONDAE2.
        USE: GAMMA.
        LOCALITY: LOCAL.
        TYPE: INTEGER.
    DATA: EXA1.
        USE: GAMMA.
        LOCALITY: LOCAL.
        TYPE: INTEGER.
    DATA: EXA2.
        USE: GAMMA.
        LOCALITY: LOCAL.
        TYPE: INTEGER.
    DATA: EXA3.
        USE: GAMMA.
        LOCALITY: LOCAL.
        TYPE: INTEGER.
    DATA: EXA4.
        USE: GAMMA.
        LOCALITY: LOCAL.
        TYPE: INTEGER.
    DATA: EXB1.
        USE: GAMMA.
        LOCALITY: LOCAL.
        TYPE: INTEGER.
    DATA: EXB2.
        USE: GAMMA.
        LOCALITY: LOCAL.
        TYPE: INTEGER.
    DATA: EXB3.
        USE: GAMMA.
        LOCALITY: LOCAL.
        TYPE: INTEGER.
    DATA: EXB4.
        USE: GAMMA.
        LOCALITY: LOCAL.
        TYPE: INTEGER.
    DATA: EXE1.
        USE: GAMMA.
        LOCALITY: LOCAL.
        TYPE: INTEGER.
```

```
      LOCALITY: LOCAL.
      TYPE: INTEGER.
DATA: EXE3.
      USE: GAMMA.
      LOCALITY: LOCAL.
      TYPE: INTEGER.
DATA: EXE4.
      USE: GAMMA.
      LOCALITY: LOCAL.
      TYPE: INTEGER.
DATA: LVB1.
      USE: GAMMA.
      LOCALITY: LOCAL.
      TYPE: INTEGER.
DATA: LVB2.
      USE: GAMMA.
      LOCALITY: LOCAL.
      TYPE: INTEGER.
DATA: ELAPA1.
      USE: GAMMA.
      LOCALITY: LOCAL.
      TYPE: REAL.
DATA: ELAPA2.
      USE: GAMMA.
      LOCALITY: LOCAL.
      TYPE: REAL.
DATA: ELAPA3.
      USE: GAMMA.
      LOCALITY: LOCAL.
      TYPE: REAL.
DATA: ELAPA4.
      USE: GAMMA.
      LOCALITY: LOCAL.
      TYPE: REAL.
DATA: ELAPB1.
      USE: GAMMA.
      LOCALITY: LOCAL.
      TYPE: REAL.
DATA: ELAPB2.
      USE: GAMMA.
      LOCALITY: LOCAL.
      TYPE: REAL.
DATA: ELAPB3.
      USE: GAMMA.
      LOCALITY: LOCAL.
      TYPE: REAL.
DATA: ELAPC1_2_3.
      USE: GAMMA.
      LOCALITY: LOCAL.
      TYPE: REAL.
DATA: ELAPC4.
      USE: GAMMA.
      LOCALITY: LOCAL.
      TYPE: REAL.
DATA: ELAPC5.
      USE: GAMMA.
```

```
      LOCALITY: LOCAL.
      TYPE: REAL.
DATA: ELAPD1.
      USE: GAMMA.
      LOCALITY: LOCAL.
      TYPE: REAL.
DATA: ELAPD2.
      USE: GAMMA.
      LOCALITY: LOCAL.
      TYPE: REAL.
DATA: ELAPE1.
      USE: GAMMA.
      LOCALITY: LOCAL.
      TYPE: REAL.
DATA: TIMEAB.
      USE: GAMMA.
      LOCALITY: LOCAL.
      TYPE: REAL.
DATA: TIMEAE1.
      USE: GAMMA.
      LOCALITY: LOCAL.
      TYPE: REAL.
DATA: TIMEAE2.
      USE: GAMMA.
      LOCALITY: LOCAL.
      TYPE: REAL.
DATA: TIMEAE3.
      USE: GAMMA.
      LOCALITY: LOCAL.
      TYPE: REAL.
DATA: TIMEAE4.
      USE: GAMMA.
      LOCALITY: LOCAL.
      TYPE: REAL.
DATA: TIMEBB1.
      USE: GAMMA.
      LOCALITY: LOCAL.
      TYPE: REAL.
DATA: TIMEBB2.
      USE: GAMMA.
      LOCALITY: LOCAL.
      TYPE: REAL.
DATA: TIMEALB1.
      USE: GAMMA.
      LOCALITY: LOCAL.
      TYPE: REAL.
DATA: TIMEALB2.
      USE: GAMMA.
      LOCALITY: LOCAL.
      TYPE: REAL.
DATA: TIMEBE.
      USE: GAMMA.
      LOCALITY: LOCAL.
      TYPE: REAL.
DATA: TIMECB1.
      USE: GAMMA.
      LOCALITY: LOCAL.
```

```
DATA: TIMECB2.
      USE: GAMMA.
      LOCALITY: LOCAL.
      TYPE: REAL.
DATA: TIMECB3.
      USE: GAMMA.
      LOCALITY: LOCAL.
      TYPE: REAL.
DATA: TIMEC13.
      USE: GAMMA.
      LOCALITY: LOCAL.
      TYPE: REAL.
DATA: TIMEC4.
      USE: GAMMA.
      LOCALITY: LOCAL.
      TYPE: REAL.
DATA: TIMECE.
      USE: GAMMA.
      LOCALITY: LOCAL.
      TYPE: REAL.
DATA: TIMEDE1.
      USE: GAMMA.
      LOCALITY: LOCAL.
      TYPE: REAL.
DATA: TIMEDB1.
      USE: GAMMA.
      LOCALITY: LOCAL.
      TYPE: REAL.
DATA: TIMEDE2.
      USE: GAMMA.
      LOCALITY: LOCAL.
      TYPE: REAL.
DATA: TIMEDB2.
      USE: GAMMA.
      LOCALITY: LOCAL.
      TYPE: REAL.
DATA: TIMEEB1.
      USE: GAMMA.
      LOCALITY: LOCAL.
      TYPE: REAL.
DATA: TIMEEB2.
      USE: GAMMA.
      LOCALITY: LOCAL.
      TYPE: REAL.
DATA: TIMEEE1.
      USE: GAMMA.
      LOCALITY: LOCAL.
      TYPE: REAL.
DATA: TIMEEE2.
      USE: GAMMA.
      LOCALITY: LOCAL.
      TYPE: REAL.

XX 002  FUNCTION RSL      COMPLETED.  **********
        SIMGEN.

XX 001  FUNCTION SIMGEN   INITIATED.  **********
```

A.2 Standardized Simulation Procedures

```
000027                          (* SETS PROCEDURES OR HEADERS   SDF          *)   00002047
000027 PROCEDURE CAUSE (ACTNO, DEVNET: INTEGER; ACTTIME: REAL);                    00002048
000006 BEGIN EXOBOOL'ACTNO] := TRUE;                                               00002049
000012   EXNOTEX'ACTNO] := TRUE;                                                   00002050
000015   EXOTIME'ACTNO] := ACTTIME;                                                00002051
000020   DEVNO'ACTNO] := DEVNET;                                                   00002052
000023   EECAUSE(EXSTR, ACTTIME)                                                   00002053
000024 END;                                                                        00002054
000042 PROCEDURE MESEUF(ACT, EXCLA, EXTYP, MESCONT: INTEGER);                      00002055
000007 BEGIN EXCLAR'ACT] := EXCLA;                                                 00002056
000013   EXTYPAR'ACT] := EXTYP;                                                    00002057
000016   MESAR'ACT] := MESCONT                                                     00002058
000020 END;                                                                        00002059
000032 PROCEDURE SSSTARTUP;                                                        00002060
000003 VAR J: INTEGER;                                                             00002061
000004 BEGIN                                                                       00002062
000004   FOR J := 1 TO 9 DO EXOBOOL'J] := FALSE;                                   00002063
000014   FOR J := 1 TO 9 DO EXNOTEX'J] := FALSE;                                   00002064
000024   FOR J := 1 TO 5 DO CURDEVS'J] := 0;                                       00002065
000034   FOR J := 1 TO 5 DO TOTDEVS'J] := 2;                                       00002066
000044   TOTDEVS'1] := 1;                                                          00002067
000045   TIMEEB1 := 0.0;                                                           00002068
000046   TIMEEB2 := 0.0;                                                           00002069
000046   EEPSTMES(EB, 0.0);                                                        00002070
000052   TIMEAB := 0.0;                                                            00002071
000053   EEPSTMES(AB, 0.0)                                                         00002072
000056 END;                                                                        00002073
000062 PROCEDURE SAE1_2;                                                           00002074
000003 BEGIN EEFSTMES(IAE1_2, FFLAG);                                              00002075
000010   TIMECB1 := TIMEAE1;                                                       00002076
000012   TIMECB2 := TIMEAE2;                                                       00002077
000013   CONDAE2 := CON02;                                                         00002078
000014   MESBUF(1,1,1,LVA2);                                                       00002079
000016   CAUSE(1, 3, TIMEAE2) ;                                                   00002080
000021   EENXTMES(IAE1_2,TRUE,FFLAG)                                               00002081
000027 END;                                                                        00002082
000032 PROCEDURE SAE3_4;                                                           00002083
000003 BEGIN EEFSTMES(IAE3_4, FFLAG);                                              00002084
000010   TIMEBB1 := TIMEAE3;                                                       00002085
000012   TIMEBB2 := TIMEAE4;                                                       00002086
000013   MESBUF(2,1,1,EXE3);                                                       00002087
000015   MESBUF(3,2,0,EXE4);                                                       00002088
000020   CAUSE(2, 2, TIMEAE3);                                                    00002089
000022   CAUSE(3, 2, TIMEAE4) ;                                                   00002090
000025   EENXTMES(IAE3_4,TRUE,FFLAG)                                               00002091
000033 END;                                                                        00002092
000036 PROCEDURE SBE;                                                              00002093
000003 BEGIN EEFSTMES(IBE, FFLAG);                                                 00002094
000010   TIMECB3 := TIMEBE;                                                        00002095
000012   MESBUF(4,2,-1,LVA7);                                                      00002096
000015   CAUSE( 4, 3, TIMEBE) ;                                                   00002097
000020   EENXTMES(IBE ,TRUE,FFLAG)                                                 00002098
000026 END;                                                                        00002099
000031 PROCEDURE SCE;                                                              00002100
000003 BEGIN EEFSTMES(ICE, FFLAG);                                                 00002101
000010   TIMEAB := TIMECE;                                                         00002102
000012   MESBUF(9,0,0,EXE3);                                                       00002103
```

```
000014    CAUSE(9, 1, TIMECE) ;                                          00002104
000017    EENXTMES(ICE ,TRUE,FFLAG)                                      00002105
000025 END;                                                              00002106
000030 PROCEDURE SDE1_2;                                                 00002107
000003 BEGIN EEFSTMES(IDE1_2, FFLAG);                                    00002108
000010    TIMEEB1 := TIMEDE1;                                            00002109
000012    TIMEEB2 := TIMEDE2;                                            00002110
000013    MESBUF(5,2,-1,LVB1);                                           00002111
000016    MESBUF(6,1,0,LVB2);                                            00002112
000021    CAUSE(5, 5, TIMEDE1);                                          00002113
000024    CAUSE(6, 5, TIMEDE2) ;                                         00002114
000027    EENXTMES(IDE1_2,TRUE,FFLAG)                                    00002115
000035 END;                                                              00002116
000040 PROCEDURE SEE;                                                    00002117
000003 BEGIN EEFSTMES(IEE, FFLAG);                                       00002118
000010    TIMEDB1 := TIMEEE1;                                            00002119
000012    TIMEDB2 := TIMEEE2;                                            00002120
000013    MESBUF(7,2,0,EXE1);                                            00002121
000015    MESBUF(8,1,0,EXE2);                                            00002122
000020    CAUSE(7, 4, TIMEEE1);                                          00002123
000023    CAUSE(8, 4, TIMEEE2) ;                                         00002124
000026    EENXTMES(IEE ,TRUE,FFLAG)                                      00002125
000034 END;                                                              00002126
000037 PROCEDURE SSEXOG;                                                 00002127
000003 VAR ACTIVE, I: INTEGER;                                           00002128
000005     DEVL, MESTEMP, K, L: INTEGER;                                 00002129
000011     MINACTTIME: REAL;                                             00002130
000012 BEGIN ACTIVE := 0;                                                00002131
000005   FOR I := 1 TO 9 DO                                              00002132
000006     IF ((EXOTIME'I] = CLOCK_TIME) AND EXNOTEX'I]) THEN ACTIVE :=I;  00002133
000023   WRITELN (OUTPUT, ' ACTIVE=', ACTIVE, CLOCK_TIME);               00002134
000041   IF ACTIVE > 0 THEN BEGIN                                        00002135
000043     K := DEVNO'ACTIVE];                                           00002136
000047     WRITELN (OUTPUT, K, CURDEVS'K], TOTDEVS'K]);                  00002137
000071     EXOBOOL'ACTIVE] := FALSE;                                     00002138
000075     EXNOTEX'ACTIVE] := FALSE;                                     00002139
000100     DEVCHK'1] := ACTIVE;                                          00002140
000101     DEVCHK'2] := 0;                                               00002141
000101     IF EXCLAR'ACTIVE] <> 0 THEN BEGIN                             00002142
000105       MINACTTIME := CLOCK_TIME;                                   00002143
000106       FOR K := 1 TO 9 DO                                          00002144
000107         IF K <> ACTIVE THEN BEGIN                                 00002145
000112           WRITELN (OUTPUT, K, EXCLAR'K], EXTYPAR'K], MESAR'K], EXCLAR' 00002146
000142               ACTIVE], EXTYPAR'ACTIVE], MESAR'ACTIVE], EXOBOOL'K],  00002147
000176               EXOTIME'K], EXNOTEX'K]);                              00002148
000215           IF (EXCLAR'K] = EXCLAR'ACTIVE]) AND ((EXTYPAR'K] = EXTYPAR' 00002149
000226               ACTIVE]) <> 1) AND (EXOTIME'K] <= MINACTTIME) AND EXOBOOL' 00002150
000240               K] THEN                                               00002151
000244             BEGIN                                                   00002152
000244             MINACTTIME := EXOTIME'K];                               00002153
000247             DEVCHK'2] := K                                          00002154
000247             END                                                     00002155
000250           END;                                                      00002156
000252       K := DEVCHK'2];                                               00002157
12 000254    IF K > 0 THEN BEGIN                                           00002158
11 000255      MESTEMP := MESAR'ACTIVE];                                   00002159
10 000261      MESAR'ACTIVE] := MESAR'K];                                  00002160
 9 000267      MESAR'K] := MESTEMP;                                        00002161
 8 000272      EXOBOOL'K] := FALSE;                                        00002162
 7 000275      EXNOTEX'K] := FALSE                                         00002163
 6 000277      END                                                         00002164
 5 000300    ELSE                                                          00002165
 4 000300      IF EXTYPAR'ACTIVE] > -1 THEN BEGIN                          00002166
 3 000305        DEVCHK'1] := 0;                                           00002167
000306           EXOBOOL'ACTIVE] := TRUE                                   00002168
000310           END                                                       00002169
000311       END;                                                          00002170
```

```
000311      FOR L := 1 TO 2 DO                                                                  00002171
000312        IF DEVCHK'LI > 0 THEN BEGIN                                                       00002172
000320          DEVL := DEVNO'DEVCHK'LII;                                                       00002173
000326          EXSCH'DEVCHK'LII := CLOCK_TIME;                                                 00002174
000334          CURDEVS'DEVLI := CURDEVS'DEVLI + 1;                                             00002175
000342          WRITELN(OUTPUT, DEVL, CURDEVS'DEVLI, TOTDEVS'DEVLI ,DEVCHK'LI)                  00002176
000372              ;                                                                           00002177
000373          IF CURDEVS'DEVLI = TOTDEVS'DEVLI THEN BEGIN                                     00002178
000402            CURDEVS'DEVLI := 0;                                                           00002179
000405            CASE DEVL OF                                                                  00002180
000411              1: BEGIN                                                                    00002181
000411                TIMEAB := EXSCH'91;                                                       00002182
000413                EEPSTMES(AB,CLOCK_TIME)                                                   00002183
000416                END;                                                                      00002184
000417              2: BEGIN                                                                    00002185
000417                TIMEBB1 := EXSCH'21;                                                      00002186
000421                TIMEBB2 := EXSCH'31;                                                      00002187
000422                EXA1 := MESAR'21;                                                         00002188
000423                EXA2 := MESAR'31;                                                         00002189
000424                EEPSTMES(BB,CLOCK_TIME)                                                   00002190
000427                END;                                                                      00002191
000430              3: BEGIN                                                                    00002192
000430                TIMECB2 := EXSCH'11;                                                      00002193
000432                TIMECB3 := EXSCH'41;                                                      00002194
000433                EXA3 := MESAR'11;                                                         00002195
000434                EXA4 := MESAR'41;                                                         00002196
000435                EEPSTMES(CB,CLOCK_TIME)                                                   00002197
000441                END;                                                                      00002198
000442              4: BEGIN                                                                    00002199
000442                TIMEDB1 := EXSCH'71;                                                      00002200
000444                TIMEDB2 := EXSCH'81;                                                      00002201
000445                EXB1 := MESAR'71;                                                         00002202
000446                EXB2 := MESAR'81;                                                         00002203
000447                EEPSTMES(DB,CLOCK_TIME)                                                   00002204
000452                END;                                                                      00002205
000454              5: BEGIN                                                                    00002206
000454                TIMEEB1 := EXSCH'51;                                                      00002207
000456                TIMEEB2 := EXSCH'61;                                                      00002208
000457                EXB3 := MESAR'51;                                                         00002209
000460                EXB4 := MESAR'61;                                                         00002210
000461                EEPSTMES(EB,CLOCK_TIME)                                                   00002211
000464                END                                                                       00002212
000465              END                                                                         00002213
000473            END                                                                           00002214
000473          END                                                                             00002215
000473      END                                                                                 00002216
000475  END;                    (* SIMULATION INITIALIZATION PROCEDURES   RISF  *)              00002217
```

```
SIMULATION OUTPUT

SIMULATOR CREATED ON 04/26/79 AT 12.59.52 WITH ID SIMULATOR/TEST_DEFAULT_ID

RUN ID: DEFAULT_SIMULATOR_RUN_ID, DATE: 04/26/79, TIME: 13.01.21
SIMULATION START TIME =                    0
SIMULATION END TIME =    1.0000000000000E+002

                    R_NET E  BEGINS TO EXECUTE - TIME =                    0
                             INPUT_INTERFACE IEB EXECUTES
                             ALPHA ALE1 EXECUTES
END ALE1         -0          0
++++++++++++
                             ALPHA ALET EXECUTES
END ALET  6.0000000000000E+000
++++++++++++
                             OUTPUT_INTERFACE IEE EXECUTES
                    R_NET E  EXECUTION ENDS

                    R_NET A  BEGINS TO EXECUTE - TIME =                    0
                             INPUT_INTERFACE IAB EXECUTES
                             ALPHA ALA1 EXECUTES
END ALA1        1001
++++++++++++
                             ALPHA ALA2 EXECUTES
END ALA2        2001         3
++++++++++++
                             ALPHA ALA1_2 EXECUTES
END ALA1_2  3.0000000000000E+000  2.0000000000000E+000
++++++++++++
                             OUTPUT_INTERFACE IAE1_2 EXECUTES
                             ALPHA ALA3 EXECUTES
END ALA3        1100
++++++++++++
                             ALPHA ALA4 EXECUTES
END ALA4        1001         1200
++++++++++++
                             ALPHA ALA3_4 EXECUTES
END ALA3_4  1.0000000000000E+000  1.0000000000000E+000
++++++++++++
                             OUTPUT_INTERFACE IAE3_4 EXECUTES
                    R_NET A  EXECUTION ENDS
ACTIVE          3  1.0000000000000E+000
          2          0          2
          1          1          1       2001          2          0       1200       TRUE  2.0000000000000E+000       TRUE
          2          1          1       1100          2          0       1200       TRUE  1.0000000000000E+000       TRUE
          4         -0         -0         -0          2          0       1200      FALSE                    0      FALSE
          5         -0         -0         -0          2          0       1200      FALSE                    0      FALSE
          6         -0         -0         -0          2          0       1200      FALSE                    0      FALSE
          7          2          0         -0          2          0       1200       TRUE  6.0000000000000E+000       TRUE
          8          1          0          0          2          0       1200       TRUE  6.0000000000000E+000       TRUE
          9         -0         -0         -0          2          0       1200      FALSE                    0      FALSE
ACTIVE          2  1.0000000000000E+000
          2          0          2
          1          1          1       2001          1          1       1100       TRUE  2.0000000000000E+000       TRUE
          3          2          0       1200          1          1       1100       TRUE  1.0000000000000E+000      FALSE
          4         -0         -0         -0          1          1       1100      FALSE                    0      FALSE
          5         -0         -0         -0          1          1       1100      FALSE                    0      FALSE
          6         -0         -0         -0          1          1       1100      FALSE                    0      FALSE
          7          2          0         -0          1          1       1100       TRUE  6.0000000000000E+000       TRUE
          8          1          0          0          1          1       1100       TRUE  6.0000000000000E+000       TRUE
          9         -0         -0         -0          1          1       1100      FALSE                    0      FALSE
```

```
ACTIVE          1  2.0000000000000E+000
          3        0          2
          2        1          1        1100          1          1        2001    TRUE  1.0000000000000E+000    FALSE
          3        2          0        1200          1          1        2001    TRUE  1.0000000000000E+000    FALSE
          4       -0         -0          -0          1          1        2001   FALSE                     0    FALSE
          5       -0         -0          -0          1          1        2001   FALSE                     0    FALSE
          6       -0         -0          -0          1          1        2001   FALSE                     0    FALSE
          7        2          0          -0          1          1        2001    TRUE  6.0000000000000E+000     TRUE
          8        1          0           0          1          1        2001    TRUE  6.0000000000000E+000     TRUE
          9       -0         -0          -0          1          1        2001   FALSE                     0    FALSE
ACTIVE          8  6.0000000000000E+000
          4        0          2
          1        1          1        2001          1          0           0    TRUE  2.0000000000000E+000    FALSE
          2        1          1        1100          1          0           0    TRUE  1.0000000000000E+000    FALSE
          3        2          0        1200          1          0           0    TRUE  1.0000000000000E+000    FALSE
          4       -0         -0          -0          1          0           0   FALSE                     0    FALSE
          5       -0         -0          -0          1          0           0   FALSE                     0    FALSE
          6       -0         -0          -0          1          0           0   FALSE                     0    FALSE
          7        2          0          -0          1          0           0    TRUE  6.0000000000000E+000     TRUE
          9       -0         -0          -0          1          0           0   FALSE                     0    FALSE
          4        1          2           8
          2        1          2           2
ACTIVE          7  6.0000000000000E+000
          4        1          2
          1        1          1        2001          2          0          -0    TRUE  2.0000000000000E+000    FALSE
          2        1          1           0          2          0          -0   FALSE  1.0000000000000E+000    FALSE
          3        2          0        1200          2          0          -0    TRUE  1.0000000000000E+000    FALSE
          4       -0         -0          -0          2          0          -0   FALSE                     0    FALSE
          5       -0         -0          -0          2          0          -0   FALSE                     0    FALSE
          6       -0         -0          -0          2          0          -0   FALSE                     0    FALSE
          8        1          0        1100          2          0          -0   FALSE  6.0000000000000E+000    FALSE
          9       -0         -0          -0          2          0          -0   FALSE                     0    FALSE
          4        2          2           7
          2        2          2           3

                   R_NET D  BEGINS TO EXECUTE - TIME =   6.0000000000000E+000
                            INPUT_INTERFACE IDB EXECUTES
                            ALPHA ALD1 EXECUTES
END ALD1       1201
++++++++++++
                            ALPHA ALD2 EXECUTES
END ALD2       1101
++++++++++++
                            ALPHA ALDT EXECUTES
END ALDT  1.6000000000000E+001  1.8000000000000E+001
++++++++++++
                            OUTPUT_INTERFACE IDE1_2 EXECUTES
                   R_NET D  EXECUTION ENDS

                   R_NET B  BEGINS TO EXECUTE - TIME =   6.0000000000000E+000
                            INPUT_INTERFACE IBB EXECUTES
                            ALPHA ALB1 EXECUTES
END ALB1          0
++++++++++++
                            ALPHA ALB2 EXECUTES
END ALB2         -0
++++++++++++
                            ALPHA ALB3 EXECUTES
ND ALB3          0
++++++++++++
                            ALPHA ALBT EXECUTES
END ALBT  1.0000000000000E+001
++++++++++++
```

```
                                OUTPUT_INTERFACE IBE EXECUTES
                          R_NET B  EXECUTION ENDS
ACTIVE          4  1.0000000000000E+001
          3        0         2
          1        1         1        2001        2        -1        -0     TRUE  2.0000000000000E+000  FALSE
          2        1         1           0        2        -1        -0    FALSE  1.0000000000000E+000  FALSE
          3        2         0          -0        2        -1        -0    FALSE  1.0000000000000E+000  FALSE
          5        2        -1        1201        2        -1        -0     TRUE  1.6000000000000E+001   TRUE
          6        1         0        1101        2        -1        -0     TRUE  1.8000000000000E+001   TRUE
          7        2         0        1200        2        -1        -0    FALSE  6.0000000000000E+000  FALSE
          8        1         0        1100        2        -1        -0    FALSE  6.0000000000000E+000  FALSE
          9       -0        -0          -0        2        -1        -0    FALSE                     0  FALSE
          3        1         2           4
ACTIVE          5  1.6000000000000E+001
          5        0         2
          1        1         1        2001        2        -1      1201     TRUE  2.0000000000000E+000  FALSE
          2        1         1           0        2        -1      1201    FALSE  1.0000000000000E+000  FALSE
          3        2         0          -0        2        -1      1201    FALSE  1.0000000000000E+000  FALSE
          4        2        -1          -0        2        -1      1201    FALSE  1.0000000000000E+001  FALSE
          6        1         0        1101        2        -1      1201     TRUE  1.8000000000000E+001   TRUE
          7        2         0        1200        2        -1      1201    FALSE  6.0000000000000E+000  FALSE
          8        1         0        1100        2        -1      1201    FALSE  6.0000000000000E+000  FALSE
          9       -0        -0          -0        2        -1      1201    FALSE                     0  FALSE
          5        1         2           5
ACTIVE          6  1.8000000000000E+001
          5        1         2
          1        1         1        2001        1         0      1101     TRUE  2.0000000000000E+000  FALSE
          2        1         1           0        1         0      1101    FALSE  1.0000000000000E+000  FALSE
          3        2         0          -0        1         0      1101    FALSE  1.0000000000000E+000  FALSE
          4        2        -1          -0        1         0      1101    FALSE  1.0000000000000E+001  FALSE
          5        2        -1        1201        1         0      1101    FALSE  1.6000000000000E+001  FALSE
          7        2         0        1200        1         0      1101    FALSE  6.0000000000000E+000  FALSE
          8        1         0        1100        1         0      1101    FALSE  6.0000000000000E+000  FALSE
          9       -0        -0          -0        1         0      1101    FALSE                     0  FALSE
          5        2         2           6
          3        2         2           1

                          R_NET E  BEGINS TO EXECUTE - TIME =   1.8000000000000E+001
                                   INPUT_INTERFACE IEB EXECUTES
                                   ALPHA ALE1 EXECUTES
END ALE1        3202       -800
++++++++++++
                                   ALPHA ALET EXECUTES
END ALET  2.4000000000000E+001
++++++++++++
                                   OUTPUT_INTERFACE IEE EXECUTES
                          R_NET E  EXECUTION ENDS

                          R_NET C  BEGINS TO EXECUTE - TIME =   1.8000000000000E+001
                                   INPUT_INTERFACE ICB EXECUTES
                                   ALPHA ALC3 EXECUTES
END ALC3        1101
++++++++++++
                                   ALPHA ALC4 EXECUTES
END ALC4           0
++++++++++++
                                   ALPHA ALC5 EXECUTES
END ALC5        1001
++++++++++++
                                   ALPHA ALCT EXECUTES
END ALCT  2.6000000000000E+001
++++++++++++
```

```
                                OUTPUT_INTERFACE ICE EXECUTES
                          R_NET C  EXECUTION ENDS
ACTIVE          8  2.4000000000000E+001
          4       0       2
          1       1       1       1101    1       0       -800    FALSE  2.0000000000000E+000   FALSE
          2       1       1          0    1       0       -800    FALSE  1.0000000000000E+000   FALSE
          3       2       0         -0    1       0       -800    FALSE  1.0000000000000E+000   FALSE
          4       2      -1         -0    1       0       -800    FALSE  1.0000000000000E+001   FALSE
          5       2      -1       1201    1       0       -800    FALSE  1.6000000000000E+001   FALSE
          6       1       0       2001    1       0       -800    FALSE  1.8000000000000E+001   FALSE
          7       2       0       3202    1       0       -800     TRUE  2.4000000000000E+001    TRUE
          9       0       0          0    1       0       -800     TRUE  2.6000000000000E+001    TRUE
ACTIVE          7  2.4000000000000E+001
          4       0       2
          1       1       1       1101    2       0       3202    FALSE  2.0000000000000E+000   FALSE
          2       1       1          0    2       0       3202    FALSE  1.0000000000000E+000   FALSE
          3       2       0         -0    2       0       3202    FALSE  1.0000000000000E+000   FALSE
          4       2      -1         -0    2       0       3202    FALSE  1.0000000000000E+001   FALSE
          5       2      -1       1201    2       0       3202    FALSE  1.6000000000000E+001   FALSE
          6       1       0       2001    2       0       3202    FALSE  1.8000000000000E+001   FALSE
          8       1       0       -800    2       0       3202     TRUE  2.4000000000000E+001   FALSE
          9       0       0          0    2       0       3202     TRUE  2.6000000000000E+001    TRUE
ACTIVE          9  2.6000000000000E+001
          1       0       1
          1       1       1          9

                    R_NET A  BEGINS TO EXECUTE - TIME =   2.6000000000000E+001
                             INPUT_INTERFACE IAB EXECUTES
                             ALPHA ALA1 EXECUTES
END ALA1        1002
++++++++++++
                             ALPHA ALA2 EXECUTES
END ALA2        1102         1
++++++++++++
                             ALPHA ALA1_2 EXECUTES
END ALA1_2  2.9000000000000E+001  2.8000000000000E+001
++++++++++++
                             OUTPUT_INTERFACE IAE1_2 EXECUTES
                             ALPHA ALA3 EXECUTES
END ALA3       -1001
++++++++++++
                             ALPHA ALA4 EXECUTES
END ALA4        1002        1201
++++++++++++
                             ALPHA ALA3_4 EXECUTES
END ALA3_4  2.7000000000000E+001  2.7000000000000E+001
++++++++++++
                             OUTPUT_INTERFACE IAE3_4 EXECUTES
                          R_NET A  EXECUTION ENDS
ACTIVE          3  2.7000000000000E+001
          2       0       2
          1       1       1       1102    2       0       1201     TRUE  2.8000000000000E+001    TRUE
          2       1       1      -1001    2       0       1201     TRUE  2.7000000000000E+001    TRUE
          4       2      -1         -0    2       0       1201    FALSE  1.0000000000000E+001   FALSE
          5       2      -1       1201    2       0       1201    FALSE  1.6000000000000E+001   FALSE
          6       1       0       2001    2       0       1201    FALSE  1.8000000000000E+001   FALSE
          7       2       0       3202    2       0       1201     TRUE  2.4000000000000E+001   FALSE
          8       1       0       -800    2       0       1201     TRUE  2.4000000000000E+001   FALSE
          9       0       0          0    2       0       1201    FALSE  2.6000000000000E+001   FALSE
          2       1       2          3
          4       1       2          7
```

Appendix B - CS-1 DPR

The following DPR example is presented in its original form with page numbers in the upper right hand corner of each page. The table of contents following the title page refers specifically to this numbering. Numbering of report pages continues uninterrupted at the bottom of each page.

DISTRIBUTED DATA PROCESSING
INTEGRATED EXPERIMENT DEFINITION

CASE STUDY 1

DATA PROCESSING REQUIREMENTS
(DPR-1)

September 1978
D.R. Fitzwater
Dept. Computer Sciences
University of Wisconsin
Madison, Wisconsin

CONTENTS

ABSTRACT

BMDATC plans to use a series of case studies to focus and integrate current DDP research. This report is a preliminary version of a DPR for CS-1. The DPR is intended to illustrate the use of asynchronously interacting processes at the DPR level of specification. This DPR is based on (and quite similar to) the TRW generated DPR for CS-1.

1. INTRODUCTION

This Data Processing Requirement (DPR) specification is based on the Required Capabilities Description (RCD) [4] for the CS-1 experiment. The contents of the RCD should be considered a part of this introduction, and only the briefest summary follows.

A baseline radar [2] as modeled by SETS [3] will be used to search and designate threatening objects to be handed over for tracking by an unspecified (outside CS-1 scope) system.

The DPR should express requirements without prejudicing subsequent DDP design as far as possible. The requirements should be unambiguous and testable to the specified level of detail.

The form of the specifications should maximize the possibility of automated analysis for desirable formal properties [5].

The following DPR is a first attempt to use asynchronously interacting processes (AIP) [5] on a real experiment. This report will illustrate a possible approach as well as providing a common formal example for clarifying the CS-1 issues. Our intent is to prepare a corresponding data processing architecture requirement (DPAR) report that will be demonstrably consistent with this DPR.

The methodology for generating and analyzing a specification such as this is presented in [5]. We will include brief, explanatory, appendices for the interpretation of exchange graphs and process specifications.

The CS-1 DPAR's in [6,7] were used (in this development) as a source of more detailed information about radar signal processing since the author has no previous experience in that area.

2. OVERVIEW

The decomposition of the CS-1 processes and the nature of their interactions is described in Figure 2.1.

The processes TS, RST, RC, and RSR represent the threat and radar sub-system as modeled by SETS [2,3]. The threat simulation (TS) drives the entire set of processes and closes the feedback loop between the radar order generator (ROG) and the radar returns process (RRP).

ROG maintains a history of the outstanding radar orders for use by RRP. There may be as many RRP processes as required to meet performance specification. RRP will analyze returns, evaluate quality, and supply detection information for futher procession.

Search return processing (SRP) will generate the first designation pulse requests for valid search detections. As many SRP can be used as are required.

Designation discrimination Set (DDS) will discriminate threatening objects via bulk filtering and a sequence of designation pulses time associated detections are stored in the TAD data base. DDS represents the set of processes described in Figure 2.2.

Known object recognition (KOR) correlates threatening objects with those already in track, and drops known objects from further processing hand-over message generation will supply any new threatening object states to the track network.

Request queue determination (RQD) will select radar requests to be scheduled in the next cycle. Radar pulse scheduling (RPS) will schedule the requests and send radar order specifications to (ROG).

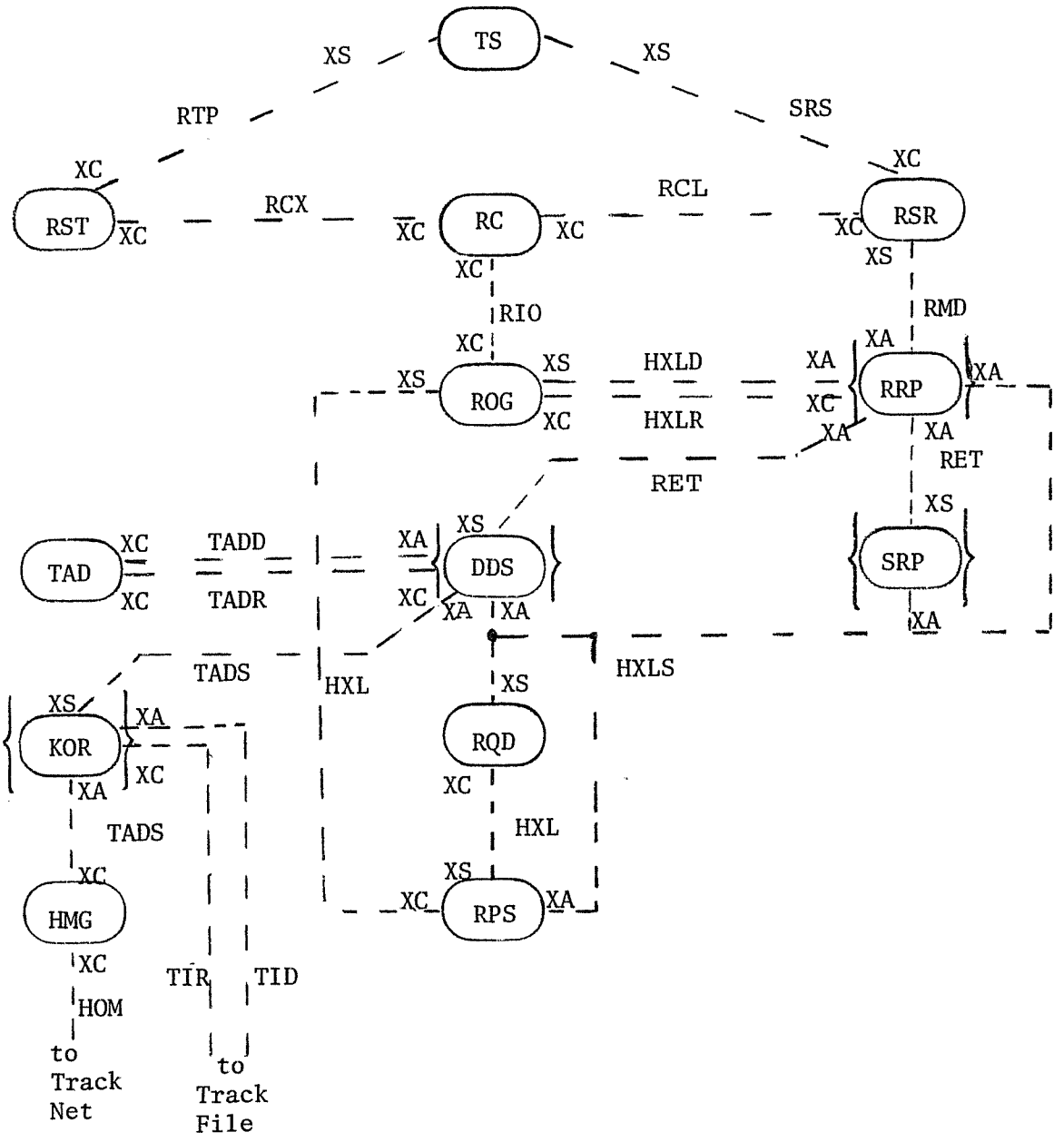


FIGURE 2-1: CS-1 DPR-1 Exchange Graph

The boxes are processes. The dashed lines represent interactions. The terminal labels identify the type of interaction (i.e., the type of exchange function used). The mid-line labels identify the message type involved in the exchange. Processes included in braces are replicated sets of identical processes. DDS is a set of processes described in Figure 2-2.

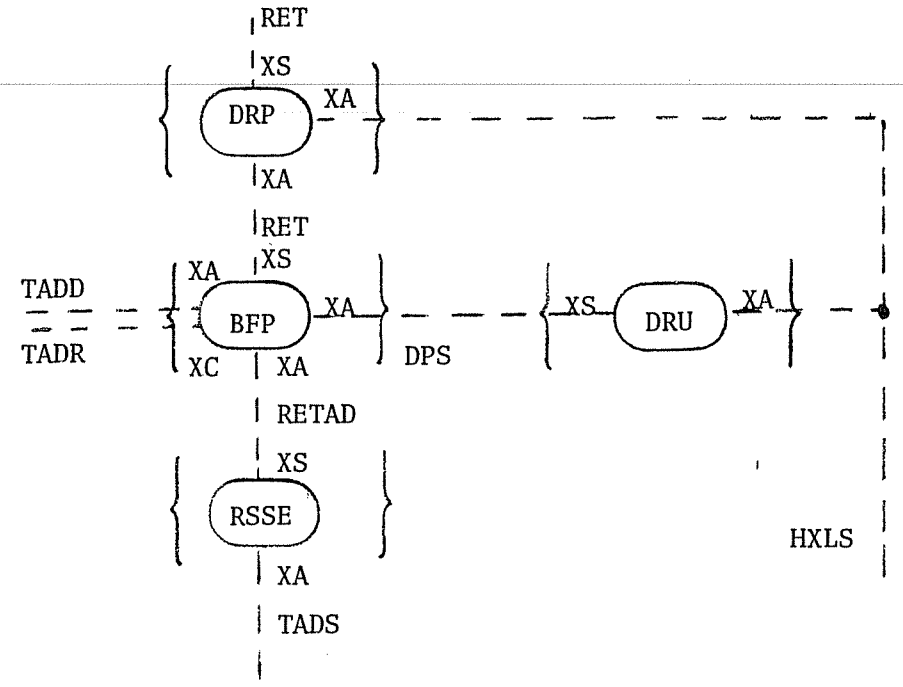


FIGURE 2-2: Exchange Graph for the replacement of DDS (See Figure 2-1) "process by" the above set of DRP, BFP, DRU, and RSSE processes.

3. PROCESSES

The general format of a process specification in this document will be as follows:

Process: Name of process

Multiplicity: Can it be replicated for performance?

Inputs: The message types that can be received in an interaction

Outputs: The message types that can be transmitted in an interaction

State: The local state components that can be referenced by the process function

Function: The defining arithmetic expression for the process state successor function. References to a state components are references to their current value in a given process step.

Square brackets in an arithmetic expression denote conditional evaluation. For example, $[P_1:E_1,P_2:E_2,E_3]$ implies that only the expression E_i associated with the first (from left to right) true predicate will be evaluated.

The protection function P_i^j selects from a j-tuple the i element. For example $P_2^3(A,B,C) = B$.

Auxiliary Functions: Functions used in defining expression for function above.

Interactions: A summary of interaction messages

Comments: English text description of process.

The following processes are arranged alphabetically.

Process: BFP

Multiplicity: Quantity sufficient

Inputs: RET

Outputs: RETAD, DPS

State: RATEINFO 'Constant State'

Function:

UBFPA(BFPA(RATEINFO,XSBFP(T))

Auxiliary Functions:

UBFPA : (RATEINFO, RETAD, DPS) → RATEINFO

P_1^3(RATEINFO,[RETAD ≠ T : XARETAD(TAD),],
[DPS ≠ T : XADPS(DPS),])

Interactions:

XSBFP : BOOLEAN → RET

XARETAD : RETAD → BOOLEAN

XADPS : DPS → BOOLEAN

Comments: Bulk Filter Processing

BFP discriminates potential objects in TAD. If this detection is null, objects that should have appeared will be dropped. Objects that do appear and cannot be discriminated as yet will produce a new designation pulse via DRU. Objects that can be declared threatening will be passed to RSSE. BFP will use and update TAD history data.

Function: BFPA : (RATEINFO, RET) → (RATEINFO, RETAD, DPS)

(Primitive)

Comments:

Implements an n-pulse non-coherent, sequential algorithm to discriminate threatening objects. Ghost and non-threatening objects will be dropped using velocity filtering based on RATEINFO.

Process: DRP

Multiplicity: Quantity sufficient

Inputs: RET

Outputs: RET, HXLS

State: Null

Function:

UDRPA(XSODD(T))

Auxiliary Functions:

UDRPA : RET → NULL

[Qual(RET) = Degraded : XARQD(DRPA(RET)), XABFP(RET)]

Interactions:

XSODD : BOOLEAN → RET

XARQD : HXLS → BOOLEAN

XATAD : TADD → BOOLEAN

XABFP : RET → BOOLEAN

Comments:

DRP will route returns according to quality. If degraded, DRP will reissue designation order. Otherwise, the return will be passed on to BFP.

Function: QUAL : RET → {Degraded, Null, Normal}

(Primitive)

Comments: Quality of return

QUAL extracts value from RET

Function: DRPA : RET → HXLS

(Primitive)

Comments: Designation Return Processing A

DRPA will regenerate a designation pulse request (HXLS) to replace the degraded return.

Function: DRPB : RET → TADD

(Primitive)

Comments: Designation Return Processing B

DRPB will form a delete order to TAD for the TAD sequence that gave rise to this detection return.

Process: DRU

Multiplicity: Quantity sufficient

Inputs: DPS

Outputs: HXLS

State: Null

Function:

XARQD(DRUA(XSDPS(T)))

Auxiliary Functions:

None

Interactions:

XSDPS : BOOLEAN → DPS

XARQD : HXLS → BOOLEAN

Comments: Designation Request Update

DRU will generate the radar request for new designation pulse.

Function: DRUA : DPS → HXLS

(Primitive)

Comments: Designation Request Update A

The designation range gates and the transmission pulse parameter specifications will be generated.

Process: HMG

Multiplicity: Unique

Inputs: TADS

Outputs: HOM

State: Null

Function:

$$P_1^2(,XCTN(HMGA(XCKOR(T))))$$

Auxiliary Function:

None

Interactions:

XCKOR : BOOLEAN → TADS

XCTN : TADS → BOOLEAN

Comments: Handover Message Generation

HMG will generate an appropriate handover message for each uncorrelated threatening object.

Function:

HMGA : TADS → HOM

(Primitive)

Comments: Handover Message Generation A

HMGA shall

a) generate state and covariance estimates of threatening object

b) generate initial track pulse information

1) beam position

2) transmit pulse length

3) allowable window

Process: KOR

Multiplicity: Quantity sufficient

Inputs: TADS

Outputs: TADS

State: Null

Function:

UKOR[KORA(XSTADS(T))]

Auxiliary Functions:

UKOR : TADS → TADS

P_1^2(,[TADS ≠ T : AKOR(TADS),])

Interactions:

XSTADS : BOOLEAN → TADS

XAKOR : TADS → BOOLEAN

Comments: Known object recognition

KOR shall attempt to correlate TADS object with a known object already in the track file for the network of which CS-1 is a part. Known objects will be dropped.

KOR uses track file with the interactions

XATID : TID → BOOLEAN and XCTIR : BOOLEAN → TIR.

Function: KORA : TADS → TADS

(Primitive)

Comments: Known Object Recognition A

KORA shall

a) correlate TADS with track file of known objects

b) terminate processing if correlated.

Process: RC

Multiplicity: Unique

Inputs: RIO

Outputs: RCL, RCX

State: RSTATUS 'Radar Status Information'

Function:

URCA(RCA(XCRIO(T),RSTATUS))

Auxiliary Function:

URCA: (RCL,RCX,RSTATUS) → RSTATUS

P_1^3(RSTATUS,[RCL ≠ T : XCRCL(RCL),]

,[RCX ≠ T : XCRCX(RCX),])

Interactions:

XCRIO : BOOLEAN → RIO

XCRCL : RCL → BOOLEAN

XCRCX : RCX → BOOLEAN

Comments: Radar Control

RC simulates radar control, accepts radar orders, and generates the appropriate transmitter (RCX) and receiver (RCL) controls. The radar status, RSTATUS, is maintained locally, and invalid orders or duty cycle conflicts will be checked. The radar to be used is specified in [2,3].

Function: RCA: (RIO,RSTATUS) → (RCL,RCX,RSTATUS)

(Primitive)

Comment: Radar Control A

RCA generates transmitter (RCX) and receiver (RCL) controls and monitors radar status (RSTATUS) to obey operational constraints. This is specified and simulated by existing CS-1 radar SETS [2,3].

Process: ROG

Multiplicity: Unique

Inputs: HXL, HXLD

Outputs: RIO, HXLR

State: ROGQ, HXLQ 'Pending Order, Outstanding Orders'

Function:

[ROGQ = T : (XSHXL(T),ROGB(HXLQ,XSHXLD(T)),
UROGA(ROGA(ROGQ,HXLQ,XSHXLD(T)))]

Auxiliary Functions:

UROGA : (ROGQ,HXLQ,RIO,HXLR) → (ROGQ,HXLQ)
P_1^3((ROGQ,HXLQ),[RIO ≠ J : XCRIO(RIO),],
[HXLR ≠ T : XCHXLR(HXLR),])

Interactions:

XSHXL : BOOLEAN → HXL

XSHXLD : BOOLEAN → HXLD

XCRIO : RIO → BOOLEAN

XCHXLR : HXLR → BOOLEAN

Comments: Radar Order Generation

ROG will generate, encode, and transmit all radar orders. ROG will also maintain a history queue of outstanding orders (whose returns have not been processed) and supply data from ROGQ to requesting RRP.

Function: ROGA : (ROGQ,HXLQ,HXLD) → (ROGQ,HXLQ,RIO,HXLR)

(Primitive)

Comments: Radar Order Generation A

ROGA shall

a) format the transmit/listen orders as RIO

b) monitor time and, when appropriate, terminate so that RIO will be transmitted

c) if there is a history demand (HXLD) evaluate ROGB.

Function: ROGB : (HXLQ,HXLD) → HXLQ

(Primitive)

Comments: Radar Order Generation B

If there is a demand (HXLD) uppdate history file HXLQ and generate HXLR response.

Process: RPS

Multiplicity: Unique

Inputs: HXL

Outputs: HXL, HXLS

State: RPSQ 'Pending Orders to be Scheduled'

Function:

URPS(RPSA(XSRP(T),RPSQ)

Auxiliary Function:

URPS : (RPSQ,HXL,HXLS) → RPSQ

P_1^3(RPSQ,[HXL ≠ T : XCHXL(HXL),],
[HXLS ≠ T : XARQD(HXLS),])

Interactions:

XSRP : BOOLEAN → HXL

XCHXL : HXL → BOOLEAN

Comments: Radar Pulse Scheduler

RPS will schedule radar pulse transmissions and corresponding receive windows in compliance with radar operational constraints. If HXL cannot be scheduled, it will be returned to RQD.

Function: RPSA : (HXL,RPSQ) → (RPSQ,HXL,HXLS)

(Primitive)

Comments: Radar Pulse Scheduler A

RPSA shall

a) subject to operational constraints the requested radar order HXL will be scheduled

b) check for overlap of transmit and receive operations and of receiver time gates

c) check for short term overlap

d) unscheduled requests will be returned to RQD for subsequent attempt.

Process: RQD

Multiplicity: Unique

Inputs: HXLS

Outputs: HXL

State: HXLQ, MAP 'Pending Radar Requests, Noise MAP'

Function:

URQDA (RQDA(HXLQ,MAP,XSRQD(I)))

Auxiliary Functions:

URQDA : (HXLQ,MAP,HXL) → (HXLQ,MAP)

P_1^2((HXLQ,MAP),[HXL ≠ T : CRP(HXL),])

Interactions:

XSRQD : BOOLEAN → HXLS

XCRP : HXL → BOOLEAN

Comments: Request Queue Determination

RQD shall establish which requests are candidates for scheduling during this radar cycle, and shall determine for all requests (search, designation) the pulse parameters to accompany the radar orders.

Function: RQDA : (HXLQ,HXLS) → (HXLQ,HXL)

(Primitive)

Comments: Radar Queue Determination A

RQDA shall

a) maintain a Queue of radar requests HXLQ

b) schedule next radar request

1) establish current cycle

2) select candidate requests

3) establish raster points and originate search requests

4) validate request parameters

5) implement search modifications

6) generate radar transmit/listen order

c) maintain a map of noise information from RRP

Process: RRP

Multiplicity: Quantity sufficient

Inputs: RMD, HXLR

Outputs: RET, HXLD, HXLS

State: (RMD, HXLR)

Function:

$$[RMD = T : URRPC(RRPC(URRPB(XARMD(T)))),$$
$$URRPA(RRPA(RMD,HXLR))]$$

Auxiliary Functions:

$$URRPA : (RMD,HXLR,RET) \rightarrow (RMD,HXLR)$$
$$P_1^2((RMD,HXLR),XAODD(RET))$$
$$URRPB : (RMD,HXLD) \rightarrow (RMD,HXLR)$$
$$(RMD,XCHXLR(XAHXLD(RRPB(RMD))))$$
$$URRPC : (RMD,HXLR,HXLS) \rightarrow (RMD,HXLR)$$
$$P_1^2((RMD,HXLR),[HXLS \neq T : XARQD(HXLS),])$$

Interactions:

$$XARMD : BOOLEAN \rightarrow RMD$$
$$XAODD : RET \rightarrow BOOLEAN$$
$$XAHXLD : HXLD \rightarrow BOOLEAN$$
$$XCHXLR : BOOLEAN \rightarrow HXLR$$
$$XARQD : HXLS \rightarrow BOOLEAN$$

Comments: Radar Returns Processing

RRP processes contend for RMD from RSR. The winning process will generate the appropriate return. RRP will assess noise content and detection quality. Detections outside of beam will be eliminated. RMD contains unprocessed detections.

Function: RRPA : (RMD,HXLR) → (RMD,HXLR,RET)

(Primitive)

Comments: Radar Return Processing A

Analyze detection for quality and then form range, angle, and range uncertainty.

Function: RRPB : RMD → (RMD,HXLD)

(Primitive)

Comments: Radar Return Processing B

RRPB simply extracts the associated radar order identification from return measurement data (RMD) and forms a request (HXLD) for relevant information (HXLR) from radar order generation.

Function: RRPC : (RMD,HXLR) → (RMD,HXLR,HXLS)

(Primitive)

Comments: Radar Return Processing C

RRPC will leave RMD, HXLR invariant and will generate HXLS for RQD with

a) order execution

b) RMD validity

c) RMD noise

d) RMD quality of radar performance.

If search return and no detections, terminate.

Suppress redundant detections.

Process: RSR

Multiplicity: Unique

Inputs: SRS, RCL

Outputs: RMD

State: Null

Function:

$$P_1^2(,URSR(RSRA(XCRCL(T))))$$

Auxiliary Functions:

URSR : RCL → BOOLEAN

XSRMD((RSRB(RCL,XCSRS(T))))

Interactions:

XCRCL : BOOLEAN → RCL

XCSRS : BOOLEAN → SRS

XSRMD : RMD → BOOLEAN

Comments: Radar Signal Reception

RSR, under control (RCL), will transform signal response (SRS) to return measurement data (RMD). The radar receiver is specified in [2,3].

Function: RSRA : RCL → RCL

(Primitive)

Comments: Radar Signal Reception A

RSRA will accept receiver control information for SRS reception. This will be specified and simulated by existing CS-1 radar SETS [2,3].

Function: RSRB : (RCL,SRS) → RMD

(Primitive)

Comments: Radar Signal Reception B

RSRB will transform signal return specifications into digital return measurements (RMD). This will be specified and simulated by existing CS-1 radar SETS [2,3].

Process: RSSE

Multiplicity: Quantity sufficient

Inputs: RETAD

Outputs: TADS

State: Null

Function:

XATADS(RSSEA(XSRETAD(T)))

Auxiliary Functions:

None

Interactions:

XSRETAD : BOOLEAN → RETAD

XATADS : TADS → BOOLEAN

Comments: Return Smoothing And State Estimation
The return and TAD sequence for a threatening object will be analyzed to produce a smoothed state estimate.

Function: RSSEA : RETAD → TADS
(Primitive)

Comments: Return Smoothing And State Estimation A
RSSE transforms measurement data from bulk filter sequences to coordinates and variances. These are then smoothed using polynomial equations. From the smoothed data, initial state and uncertainty estimate are derived.

Process: RST

Multiplicity: Unique

Inputs: RCX

Outputs: RTP

State: Null

Function:

$$P_1^2(,XCRTP(RSTA(XCRCX(T))))$$

Auxiliary Function:

None

Interactions:

XCRCX : BOOLEAN → RCX

XCRTP : RTP → BOOLEAN

Comment: Radar Signal Transmission

RST simulates a radar transmitter and generates an RTP subject to the controls contained in RCX. The radar transmitter is specified in [2,3].

Function: RSTA : RCX → RTP

(Primitive)

Comment: Radar Signal Transmission A

RSTA generates a radar pulse (RTP) as specified by transmitter control (RCX) information. This will be specified and simulated by the existing CS-1 radar SETS [2,3].

Process: SRP

Multiplicity: Quantity sufficient

Inputs: RET

Outputs: HXLS

State: Null

Function:

USRP(SRPA(XSODD(T)))

Auxiliary Functions:

USRP : HXLS → Null

P_1^2(,[HXLS ≠ T : XARQD(HXLS),])

Interactions:

XSODD : BOOLEAN → RET

XARQD : HXLS → BOOLEAN

Comments: Search Return Processing

SRP processes will contend for RET from RRP. The winning process will reassess the quality of the detection. Invalid detections will be ignored, however pulse parameter modifications for the next scan may be requested. Valid detections will cause the generation of a radar command for the first designation pulse.

Function: SRPA : RET → (RET,HXLS)

(Primitive)

Comments: Search Return Processing A

a) for degraded returns, determine pulse modifications and notify RQD via HXLS

b) for nominal returns insure that range, angle, and range uncertainty estimates for the detection are valid

c) for nominal returns specify the first designation pulse and corresponding range gates in HXLS

Process: TAD

Multiplicity: Unique

Inputs: TADD

Outputs: TADR

State: TADV 'TAD History'

Function:

UTAD(TADA(TADV,XCTADD(T)))

Auxiliary Functions:

UTAD : (TADV,TADR) → TADV

P_1^2(TADV,[TADR ≠ T : XCTADR(TADR),])

Interactions:

XCTADD : BOOLEAN → TADD

XCTADR : TADR → BOOLEAN

Comments: Time Associated Detections

TADV is a set of detection sequences that potentially describe an object. TAD is a data base process that will supply detection history information to the requesting designation return process.

Function: TADA : (TADV,TADD) → (TADV,TADR)

(Primitive)

Comments: Time Associated Data A

The current file of active time associated detection data (to be used by DRP) is maintained by processing a request (TADD) against the current file (TADV) and producing an updated file and a response (TADR).

Process: TS

Multiplicity: Unique

Inputs: RTP

Outputs: SRS

State: (TSS,RTPA) 'Threat Status, Generated Pulse Queue'

Function:

(TSA(TSS), UTSB (TSB (XSRTP (T),RTPQ,TSS)))

Auxiliary Functions:

UTSB : (RTPQ,SRS) → RTPQ

P_1^2(RTPQ,XSSRS(SRS))

Interactions:

XSSRS : SRS → BOOLEAN

XSRTP : BOOLEAN → RTP

Comments: Threat Simulator.

TS simulates the threat (as defined in RCD[1])and processes radar pulses to form signal returns. TSS defines the current threat status. RTPQ is the queue of the radar pulses for which returns are still pending. The step time of TS will be a "constant" that defines a unit of real time. TS is thus a clocking process and does not wait to interact.

Function: TSA : TSS → TSS

(Primitive)

Comments: Threat Simulation A

TSS defines threat status change for one time unit. The threat is described in RCD [1]. This will be specified and simulated by the existing CS-1 radar SETS [2,3].

Function: TSB : (RTP,RTPQ,TSS) → (RTPQ,SRS)

(Primitive)

Comments: Threat Simulation B

Given the current threat status (TSS), the pending radar pulse specifications (RTPQ), and a new radar pulse (if any), TSB will update RTPQ and generate the appropriate signal return specifications (SRS) for transmission to radar receiver (RSR). This will be specified and simulated by the existing CS-1 radar SETS [2,3].

4. MESSAGES

The general format for a message specification in this report is as follows:

MSG: Name of message type

ATTRIBUTES: (ATTR.1,...,ATTR.n) where each attribute is a message type or a value name.

VALUES: A list of value names (local to this message type) and their description.

COMMENTS: English text description of message type.

MSG: DPS

ATTRIBUTES: (Primitive)

COMMENTS: Designation pulse specification

DPS will contain all information required for DRU to determine appropriate HXLS.

MSG: HOM

ATTRIBUTES: (Primitive)

COMMENTS: Hand over message

HOM will contain an estimated target state and uncertainty. HOM will also contain beam position, transmit pulse length, and allowable transmission windows for the initial track pulse.

MSG: HXL

ATTRIBUTES: (Primitive)

COMMENTS: History of transmit/listen

HXL will contain the information required by the radar pulse scheduler for a transmit/listen command.

MSG: HXLD

ATTRIBUTES: (Primitive)

COMMENTS: History data demand

HXLD requests radar orders from ROG for associating with returns by RRP.

MSG: HXLR

ATTRIBUTES: (Primitive)

COMMENTS: History data response

HXLR contains radar order information requested by a preceding HXLD.

MSG: HXLS

ATTRIBUTES: (RRP,HXLR,RCONF,VALIDITY,NOISE,QUALITY)
(SRP,SPM)
(XL,BIC,TYPE,ET,TPD,RGD,RCP,RMGCP)

VALUES:

RRP,SRP,XL = MESSAGE SUB-TYPES

BIC = BEAM IDENTIFICATION CODE

TYPE = OF RADAR REQUEST

ET = DESIRED PULSE TIME

TPD = TRANSMIT PULSE DURATION

RGD = RANGE GATE DURATION

RCP = RADAR CONTROL PARAMETERS

RMGCP = RMG CONTROL PARAMETERS

SPM = SEARCH PULSE MODIFICATIONS

MSG: SRS

ATTRIBUTES: (Primitive)

COMMENTS: Signal return specification

SRS is a specification of a radar signal return, that will be interpreted by the radar receiver RSR.

MSG: RET

ATTRIBUTES: (HXLS,AVENOS,QUAL,AMPL,RMRNG,ACOOR,ERRS)

VALUES:

HXLS = ORIGINATING RADAR ORDER

AVENOS = AVERAGE VALUE OF VIDEO NOISE

QUAL = NULL, DEGRADED, GOOD

AMPL, ACOOR = SEE RMD

RMRNG = RANGE MARK RANGE

ERRS = ERROR ESTIMATES FOR RMRNG, ACOOR.

COMMENTS:

Return defines a radar order and a subsequent detection for further processing.

MSG: RETAD

ATTRIBUTES: (RET,TADR)

COMMENTS: Return and Time associated data

RETAD summarizes data from bulk filter on a threatening object.

MSG: RMD

ATTRIBUTES: (LNAME, RRCONF, RRBOF, AVENOS, NRMS, DET,..,DET)

VALUES:

LNAME = NAME OF ORIGINATING LISTEN ORDER

RRCONF = TRUE IF ORDER WAS EXECUTED

RRBOF = TRUE IF MORE THAN MAXIMUM DETECTIONS

AVENOS = AVERAGE VALUE OF VIDEO NOISE

NRMS = NUMBER OF RANGE MARKS

DET = (AMPL, RMT, ACOOR)

AMPL = AMPLITUDE OF RANGE MARK VIDEO

RMT = RANGE MARK TIME

ACOOR = ANGULAR COORDINATES RELATIVE TO BEAM CENTER

COMMENTS: Return measurement data

A radar order (LNAME) has produced a set of detections whose digital characterization is defined by RMD.

MSG: RTP

ATTRIBUTES: (Primitive)

COMMENTS: Radar transmitted pulse

RTP specifies a radar pulse that will be interpreted by the threat simulator (TS) to produce appropriate signal returns (SRS).

MSG: TADD

ATTRIBUTES: (Primitive)

COMMENTS: Time associated data demand

TADD will be interpreted by TAD to request or store TADV state data.

MSG: TADR

ATTRIBUTES: (Primitive)

COMMENTS: Time associated data response

If a previous TADD requires a response, TADR will contain the requested TADV data.

MSG: TADS

ATTRIBUTES: (Primitive)

COMMENTS: Time associated detection sequence

TADS will contain the smoothed state estimation of a discriminated threatening object.

5. PERFORMANCES

[TBS]

NOTE: The performance requirements in the TRW DPR could be considered part of this specification. We have not had time to do a performance analysis of this CS-1 DPR.

REFERENCES

[1] DDP Integrated Experiment Definition, Case Study 1, Required Capability Description (RCD) TRW Report No. 32304-6921-003, 21 March 1978.

[2] Baseline Radar Interface Performance Specification, Systems Development Corporation, Doc. No. 22944-9765-LE00-001, 8 January 1974.

[3] Baseline SETS IRTSW Interface Requirements Specification, Systems Development Corporation, Report No. TM-HU-144/000/01, 11 November 1974.

[4] DDP Integrated Experiment Definition, Case Study 1, Data Processing Requirements (DPR), TRW Report No.32304-6921-004, 21 March 1978.

[5] The Formal Design and Analysis of Distributed Data-Processing Systems, Univ. of Wisconsin-Madison, CSTR 322, April 1978.

[6] DDP Integrated Experiment Definition, Case Study 1, Distributed Architecture Requirements (DAR), TRW Report No. 32304-6921-005, 21 March 1978.

[7] Distributed Processing Architecture Requirements for Case Study 1, GRC Technical Report No. IM-2170 (Draft) August 1978.

APPENDIX C: Equi-Phase Simulation Example.

We can define a three process system that includes a real time clock, a user process, and an off-loaded, shared transformation process as:

```
clk(n)=P12([n<CLKMAX:n+1,0],xiCKT(n))
g(n)=P12(n+1,xcGR(sum(xcGD())))
f(n1,n2)=(xcGR(xrGD((n1,xrCKT()))),
          xcGR(xrGD((n2,xrCKT())))).
```

This system consists of a "real time" clock clk, a computational process f, and an off-loaded function process g. The g process simply accepts a two component message, adds the components, and returns the result to the requesting process, while counting the number of times it has been invoked. The real time clock ticks to CLKMAX and recycles to zero, while providing a current time message to any requesting process. The f process simply carries two independent sums involving times of process steps.

This non-trivial, but small system was translated into the multi-tasking code below. The simplicity, brevity, and clarity of the AIP form is clearly demonstrated by the equivalent program below.

1. Multi-Tasking Program.

```
te1: ;
     oinit(3,"((i)(i)(ii))");
     goto sched;
te3:;
     tinit(2,6);
     goto sched;
te7: ;
     if(d[par(1)]<CLKMAX){rval(1);
                          ++d[o[task]];
                          goto te9;
                          }
     else{d[o[task]]=0;
          t[o[task]]=IN;
          }
te9: ;
     stuff(1); goto te6;
te8: ;
     rval(1);
     dot(10);
     val=XI: tmp=CKT; goto exchange;
te10: ;
     stuff(2);
te6: ;
     if(q[task]) goto sched;
     oldlc(remove(res(2)));
     if(t[o[task]]!=NL){supdt(1); goto te3;}
     stuff(1); goto te2;
te4: ;
     tinit(2,11); goto sched;
te12: ;
     rval(1);
     ++d[o[task]];
     stuff(1); goto te11;
te13: ;
```

```
      dot(14);
      val=XC; tmp=GD; goto exchange;
te14: ;
      dot(15);
      goto sum;
te15: ;
      dot(16);
      val=XC; tmp=GR; goto exchange;
te16: ;
      stuff(2); goto te11;
te11: ;
      if(q[task]) goto sched;
      oldlc(remove(res(2)));
      if(t[p[task]]!=NL){supdt(2); goto te4;}
      stuff(2); goto te2;
te5: ;
      tinit(2,17); goto sched;
te18: ;
      tinit(2,20); goto sched;
te21: ;
      rval(1);
      stuff(1); goto te20;
te22: ;
      dot(23);
      val=XR; tmp=CKT; goto exchange;
te23: ;
      stuff(2);
te20: ;
      if(q[task]) goto sched;
      dot(24);
      val=XR; tmp=GD; goto exchange;
te24: ;
      dot(25);
      val=XC; tmp=GR; goto exchange;
te25: ;
      stuff(1); goto te17;
te19: ;
      tinit(2,26); goto sched;
te27: ;
      rval(2);
      stuff(1); goto te26;
te28: ;
      dot(29);
      val=XR; tmp=CKT; goto exchange;
te29: ;
      stuff(2);
te26: ;
      if(q[task]) goto sched;
      dot(30);
      val=XR; tmp=GD; goto exchange;
te30: ;
      dot(31);
      val=XC; tmp=GR; goto exchange;
te31: ;
      stuff(2); goto te17;
te17: ;
      if(q[task]) goto sched;
      if(t[p[task]]!=NL){supdt(3); goto te5;}
      stuff(3); goto te2;
te2: ;
      if(q[task]) goto sched;
```

The program above was then combined with the simulator program and compiled into executable code.

2. Equi-Phase Simulation Results.

The compiled system specification was then executed in the equi-phase simulator to produce several types of analysis data, as shown in the table below of edited simulator output. The first column traces the process state values in their completion sequence. The second column traces the task execution sequence. The third column traces the exchange function interactions in their initiation sequences for each of the phase steps.

The columns are correlated by phase step numbers. A phase consists of the execution of all pending tasks until no more can be done without interactions between the tasks (exchange function completions). Then all possible interactions are completed, and a new phase is initiated. Thus, the relative rates of execution are controled by the interaction frequencies of the processes.

The initial process state values were all set to zero in this simulation run. Of course, any other values could have been used.

Process State	Task No.	Exchange Type Class Ret.
done step 1	5	done step 1
1 (1)	4	2, 3, 10
done step 2	3	1, 3, 29
1 (2)	19	1, 3, 23
done step 3	18	3, 5, 14
1 (3)	13	done step 2
2 (1)	12	1, 3, 23
done step 4	8	2, 3, 10
1 (4)	7	3, 5, 14
done step 5	28	1, 5, 30
1 (5)	27	done step 3
3 (1,0)	22	2, 3, 10
2 (2)	21	3, 4, 16
done step 6	done step 1	3, 4, 31
1 (6)	0	1, 5, 24
done step 7	0	done step 4
1 (7)	8	2, 3, 10
done step 8	7	1, 5, 24
1 (8)	done step 2	3, 5, 14
2 (3)	0	done step 5
done step 9	0	2, 3, 10
1 (9)	0	3, 4, 25
done step 10	0	3, 4, 16
1 (10)	8	done step 6
3 (7,5)	7	2, 3, 10
2 (4)	done step 3	1, 3, 29
done step 11	0	1, 3, 23
1 (0)	0	3, 5, 14

```
done step 12
 1     (1)
done step 13
 1     (2)
 2     (5)
done step 14
 1     (3)
done step 15
 1     (4)
 3     (7,15)
 2     (6)
done step 16
 1     (5)
done step 17
 1     (6)
done step 18
 1     (7)
 2     (7)
done step 19
 1     (8)
done step 20
 1     (9)
 3     (12,19)
 2     (8)
done step 21
 1     (10)
done step 22
 1     (0)
done step 23
 1     (1)
 2     (9)
done step 24
 1     (2)
done step 25
 1     (3)
 3     (22,28)
 2     (10)
done step 26
 1     (4)
done step 27
 1     (5)
done step 28
 1     (6)
 2     (11)
done step 29
 1     (7)
done step 30
 1     (8)
 3     (26,31)
 2     (12)
done step 31
 1     (9)
done step 32
 1     (10)
done step 33
 1     (0)
 2     (13)
done step 34
 1     (1)
done step 35
 1     (2)
```

```
0
8
7
13
12
done step 4
0
0
0
8
7
done step 5
0
0
0
8
7
19
18
13
12
28
27
22
21
done step 6
0
0
8
7
done step 7
0
0
0
0
8
7
done step 8
0
0
0
8
7
13
12
done step 9
0
0
0
8
7
done step 10
0
0
0
8
7
19
18
13
12
```

```
done step 7
 1,    3,   23
 2,    3,   10
 3,    5,   14
 1,    5,   30
done step 8
 2,    3,   10
 3,    4,   16
 3,    4,   31
 1,    5,   24
done step 9
 2,    3,   10
 1,    5,   24
 3,    5,   14
done step 10
 2,    3,   10
 3,    4,   25
 3,    4,   16
done step 11
 2,    3,   10
 1,    3,   29
 1,    3,   23
 3,    5,   14
done step 12
 1,    3,   23
 2,    3,   10
 3,    5,   14
 1,    5,   30
done step 13
 2,    3,   10
 3,    4,   16
 3,    4,   31
 1,    5,   24
done step 14
 2,    3,   10
 1,    5,   24
 3,    5,   14
done step 15
 2,    3,   10
 3,    4,   25
 3,    4,   16
done step 16
 2,    3,   10
 1,    3,   29
 1,    3,   23
 3,    5,   14
done step 17
 1,    3,   23
 2,    3,   10
 3,    5,   14
 1,    5,   30
done step 18
 2,    3,   10
 3,    4,   16
 3,    4,   31
 1,    5,   24
done step 19
 2,    3,   10
 1,    5,   24
 3,    5,   14
done step 20
```

```
 3     (35,39)
 2     (14)
done step 36
 1     (3)
done step 37
 1     (4)
done step 38
 1     (5)
 2     (15)
done step 39
 1     (6)
done step 40
 1     (7)
 3     (39,41)
 2     (16)
done step 41
 1     (8)
done step 42
 1     (9)
done step 43
 1     (10)
 2     (17)
done step 44
 1     (0)
done step 45
 1     (1)
 3     (46,48)
 2     (18)
done step 46
 1     (2)
done step 47
 1     (3)
done step 48
 1     (4)
 2     (19)
done step 49
 1     (5)
done step 50
 1     (6)
 3     (48,49)
 2     (20)
done step 51
 1     (7)
done step 52
 1     (8)
done step 53
 1     (9)
 2     (21)
done step 54
 1     (10)
done step 55
 1     (0)
 3     (55,55)
 2     (22)
done step 56
```

```
  28
  27
  22
  21
done step 11
   0
   0
   8
   7
done step 12
   0
   0
   0
   8
   7
done step 13
   0
   0
   0
   8
   7
  13
  12
done step 14
   0
   0
   0
   8
   7
done step 15
   0
   0
   0
   8
   7
  19
  18
  13
  12
  28
  27
  22
  21
done step 16
   0
   0
   8
   7
done step 17
   0
   0
   0
   0
   8
```

```
 2,    3,    10
 3,    4,    25
 3,    4,    16
done step 21
 2,    3,    10
 1,    3,    29
 1,    3,    23
 3,    5,    14
done step 22
 1,    3,    23
 2,    3,    10
 3,    5,    14
 1,    5,    30
done step 23
 2,    3,    10
 3,    4,    16
 3,    4,    31
 1,    5,    24
done step 24
 2,    3,    10
 1,    5,    24
 3,    5,    14
done step 25
 2,    3,    10
 3,    4,    25
 3,    4,    16
done step 26
 2,    3,    10
 1,    3,    29
 1,    3,    23
 3,    5,    14
done step 27
 1,    3,    23
 2,    3,    10
 3,    5,    14
 1,    5,    30
done step 28
 2,    3,    10
 3,    4,    16
 3,    4,    31
 1,    5,    24
done step 29
 2,    3,    10
 1,    5,    24
 3,    5,    14
done step 30
 2,    3,    10
 3,    4,    25
 3,    4,    16
done step 31
 2,    3,    10
 1,    3,    29
 1,    3,    23
 3,    5,    14
done step 32
```

3. Conclusions.

The equi-phase simulation of AIP specifications of system requirements is fast and easily exploited to conduct analysis experiments on the behavior of the specified system. This type of design feedback is practical at any level of the development process. Further, this is just one of several possible modes of specification interpretation.