DESIGN TOOLS FOR EVALUATING MULTIPROCESSOR PROGRAMS

CARNEGIE-MELLON UNIVERSITY
PITTSBURGH, PENNSYLVANIA

JULY 1976

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER: AFOSR - TR - 77 - 0015	2. GOVT ACCESSION NO.
3. RECIPIENT'S CATALOG NUMBER	
4. TITLE (and Subtitle): DESIGN TOOLS FOR EVALUATING MULTIPROCESSOR PROGRAMS	5. TYPE OF REPORT & PERIOD COVERED: Interim
	6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s): Philip Howard Mason	8. CONTRACT OR GRANT NUMBER(s): F44620-73-C-0074
9. PERFORMING ORGANIZATION NAME AND ADDRESS: Carnegie-Mellon University, Computer Science Dept., Pittsburgh, PA 15213	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS: 61101D, AO 2466
11. CONTROLLING OFFICE NAME AND ADDRESS: Defense Advanced Research Projects Agency, 1400 Wilson Blvd, Arlington, VA 22209	12. REPORT DATE: July 1976
	13. NUMBER OF PAGES: 203
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office): Air Force Office of Scientific Research (NM), Bolling AFB, DC 20332	15. SECURITY CLASS. (of this report): UNCLASSIFIED
	15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited

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

18. SUPPLEMENTARY NOTES

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

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

An approach to designing programs for implementation in a multiple instruction stream-multiple data stream processing environment is presented. A program is modeled as a directed graph consisting of two types of nodes: processing nodes and linking nodes. Communication among nodes in the model is represented by message tokens. Each processing node is similar in form to a semi-Markov process. A simulation of the operation of the model is nondeterministic, but is based on prescribed probabilisti

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

DISCLAIMER NOTICE

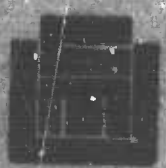
THIS DOCUMENT IS THE BEST QUALITY AVAILABLE.

COPY FURNISHED CONTAINED A SIGNIFICANT NUMBER OF PAGES WHICH DO NOT REPRODUCE LEGIBLY.

Design Tools for Evaluating Multiprocessor Programs

Philip Howard Mager

DEPARTMENT
of
COMPUTER SCIENCE



Carnegie-Mellon University

choice functions. A system, called STEPPS, has been built in which a model can be described and evaluation tools can be used to manipulate and act upon a model to predict performance of a program decomposition.

The design approach is to describe a multiprocessing program in terms of the modeling system. The model is examined to determine some analytic attributes of the model. The analysis available determines (a) whether the model is well formed, (b) whether the model contains deadlocks, (c) predictions of steady state properties of each process. In addition, without much difficulty, analysis functions external to STEPPS may be included as needed by a program designer.

Some analyses, that may be interesting, may be difficult to determine without resorting to simulation. Therefore the STEPPS system includes a model simulator with data collection facilities. The STEPPS data collection facilities include such measures as wait times and queue lengths. As in the case of analysis functions, STEPPS allows the inclusion of data collection facilities not originally provided by STEPPS.

As a system is designed, alternate models can be examined; and based on an individual designer's choice of performance attributes, a model can be chosen on which to base the construction of a multiprocessor program. As more is learned about the real system parameters, the model can be tuned to more closely predict ultimate system performance.

Several examples of communicating processes are modeled using STEPPS including pipeline processes, probabilistic processes, P/V synchronization, and reader/writer synchronization. Two experiments are presented as validation of the usefulness of the STEPPS tools. In the Bliss/11 experiment, the implications of restricting the numbers of available processors and using different scheduling algorithms were examined, and the effect of using alternate program structures was explored. In the Hearsay II experiment it was shown that, when a multiprocess program under development is sufficiently instrumented, the STEPPS model and system can be used to help tune the program's structure.

The use of the tools for predicting the performance of a multiprocessing program falls between purely analytic models, such as queueing theory or Petri-nets, and system simulations built in a general purpose simulation language. The STEPPS system is presented as a new approach to designing multiprocessing programs.

Design Tools for Evaluating Multiprocessor Programs

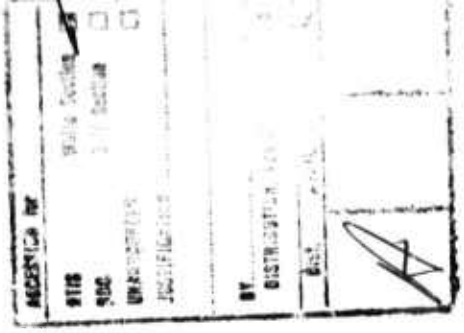
Philip Howard Mason

Department of Computer Science
Carnegie-Mellon University
Pittsburgh, Pa. 15213
July, 1976

Submitted to Carnegie-Mellon University in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

This research was supported by the Defense Advanced Research Projects Agency of the Office of the Secretary of Defense (F44620-73-C0074) and is monitored by the Air Force Office of Scientific Research.

(c)



Abstract

An approach to designing programs for implementation in a multiple instruction stream-multiple data stream processing environment is presented. A program is modeled as a directed graph consisting of two types of nodes: processing nodes and linking nodes. Communication among nodes in the model is represented by message tokens. Each processing node is similar in form to a semi-Markov process. A simulation of the operation of the model is nondeterministic, but is based on prescribed probabilistic choice functions. A system, called STEPPS, has been built in which a model can be described and evaluation tools can be used to manipulate and act upon a model to predict performance of a program decomposition.

The design approach is to describe a multiprocessing program in terms of the modeling system. The model is examined to determine some analytic attributes of the model. The analysis available determines (a) whether the model is well formed, (b) whether the model contains deadlocks, (c) predictions of steady state properties of each process. In addition, without much difficulty, analysis functions external to STEPPS may be included as needed by a program designer.

Some analyses, that may be interesting, may be difficult to determine without resorting to simulation. Therefore the STEPPS system includes a model simulator with data collection facilities. The STEPPS data collection facilities include such measures as wait times and queue lengths. As in the case of analysis functions, STEPPS allows the inclusion of data collection facilities not originally provided by STEPPS.

As a system is designed, alternate models can be examined; and based on an individual designer's choice of performance attributes, a model can be chosen on which to base the construction of a multiprocessor program. As more is learned about the real system parameters, the model can be tuned to more closely predict ultimate system performance.

Several examples of communicating processes are modeled using STEPPS including pipeline processes, probabilistic processes, P/V synchronization, and reader/writer synchronization. Two experiments are presented as validation of the usefulness of the STEPPS tools. In the Bliss/11 experiment, the implications of restricting the numbers of available processors and using different scheduling algorithms were examined, and the effect of using alternate program structures was explored. In the Hearsay II experiment it was shown that, when a multiprocess program under development is sufficiently instrumented, the STEPPS model and system can be used to help tune the program's structure.

The use of the tools for predicting the performance of a multiprocessing program falls between purely analytic models, such as queueing theory or Petri-nets, and system simulations built in a general purpose simulation language. The STEPPS system is presented as a new approach to designing multiprocessing programs.

ACKNOWLEDGEMENT

I sincerely thank my thesis committee who with their advice, guidance, and criticism of this thesis helped me to maintain their high standards: Bill Wulf (chairman), Sam Fuller, Charles Kriebel, Victor Lesser, and Mary Shaw. I am grateful for having been associated with the Carnegie-Mellon Computer Science Department and I must acknowledge the initial, and continuing, inspiration gleaned from Alan J. Perlis, my first computer science teacher, former Carnegie-Mellon department head and supervisor. In addition, I am grateful for the interest, support, and assistance from my friends, colleagues, family, and especially my parents.

Most of all, I thank my wife, Lee, for suffering through all the lonely nights (and days), for helping me to rewrite many pages, for learning to use the computer to type this thesis, for keeping me going, and for her understanding.

TABLE OF CONTENTS

CHAPTER			*PAGE*
I	Problem Statement, History and Goals		
	I.A	Introduction	I-1
	I.B	Direction of this work	I-5
	I.C	Other work bearing on the problem	I-8
	I.D	The STEPPS System	I-15
	I.E	The STEPPS system and simulator	I-24
	I.F	Thesis contributions and outline of remainder of thesis	I-29
II	The STEPPS Model		
	II.A	Modeling the behavior of a process	II-1
	II.B	Data flow and links	II-3
	II.C	Notation and definitions	II-6
	II.D	STEPPS system capabilities	II-11
III	The Use of the STEPPS Approach to Program Design		
	III.A	Use of the STEPPS model	III-1
	III.B	Using STEPPS during system design: A Bliss/11 compiler	III-12
	III.C	Using STEPPS during system construction and tuning: Hearsay II	III-28
IV	Analysis of a STEPPS Model		
	IV.A	Markov and semi-Markov processes	IV-1
	IV.B	Well-formed STEPPS models	IV-6
	IV.C	Deadlock structures and situations	IV-8
	IV.D	Reducing a STEPPS model	IV-13

IV.E The recognition of deadlocks IV-29

V The STEPPS Simulator and STEPPS Interactive System

V.A Simulation objectives V-1

V.B Simulation operation and data collected V-3

V.C The implementatiion of the STEPPS system V-10

VI Summary

VI.A Designing programs for multiprocessor computers VI-2

VI.B Experiments and results VI-5

VI.C Future research and refinements to STEPPS VI-9

VI.D Conclusions VI-12

A STEPPS System Manual

A.1 Introduction A-1

A.2 Model creation A-3

A.3 Model analysis and system commands A-8

A.4 Simulation commands A-8

A.5 Keyword commands A-9

B Using the STEPPS System

B.1 Bliss/11 example protocol B-1

B.2 The STEPPS Hearsay II model B-3

C Validation of Simulation Results

Bibliography i

Index vii

FIGURES

FIGURE		PAGE
Figure I-1	Possible relationships between two processes, A and B	I-5
Figure I-2	A marked graph	I-10
Figure I-3	A finite state atomaton	I-10
Figure I-4	A Petri net that is neither a marked graph nor a finite state atomaton	I-11
Figure I-5	UCLA model nodes	I-13
Figure I-6	Pipeline	I-17
Figure I-7	Registrar's data retrieval system	I-20
Figure I-8	Process ALPHA	I-21
Figure I-9	Mapping between Petri nets and STEPPS model	I-23
Figure I-10	Mapping of UCLA model to STEPPS	I-24
Figure I-11	Incompatible loop	I-27
Figure I-12	Incompatible non-loop	I-27
Figure II-1	Process and link graphical notation	II-8
Figure III-1	Fork and join processes	III-2
Figure III-2	Subroutine process	III-4
Figure III-3	Concurrent processing subroutine call	III-5
Figure III-4	Poisson arrival process	III-6
Figure III-5	General service time process	III-7
Figure III-6	Pipeline of processes	III-8
Figure III-7	Lock/Unlock synchronization	III-10
Figure III-8	Reader/Writer synchronization	III-11
Figure III-9	Bliss/11 phase structure	III-13

Figure III-10	Bliss/11 measured data	III-17
Figure III-11	STEPPS Bliss/11 model commands	III-17
Figure III-12	Bliss/11 graph model	III-18
Figure III-13	Bliss/11 simulation FIFO table	III-18
Figure III-14	Bliss/11 simulation LINK table	III-19
Figure III-15	Bliss/11 simulation RANDOM table	III-19
Figure III-16	Bliss/11 percentage maximum throughput	III-20
Figure III-17	Graph of measured throughput	III-21
Figure III-18	Bliss/11 simulation FIFO queue lengths	III-22
Figure III-19	Bliss/11 simulation LINK queue lengths	III-23
Figure III-20	Bliss/11 simulation RANDOM queue lengths	III-23
Figure III-21	Table of results of multi-copy Bliss/11 phase models	III-24
Figure III-22	Multi-copy Bliss/11 phase model Thru Rate graph	III-25
Figure III-23	Multi-copy Bliss/11 phase model percentage Max Thru Rate graph	III-26
Figure III-24	LEX decomposition results	III-27
Figure III-25	Simplified HSII system organization	III-32
Figure III-26	Description of precondition process	III-33
Figure III-27	STEPPS precondition model	III-33
Figure III-28	Knowledge Source process description	III-34
Figure III-29	STEPPS Knowledge Source model	III-35
Figure III-30	PCSELECTOR process	III-35
Figure III-31	Set of identical Knowledge Sources	III-36
Figure III-32	Hearsay II locking structure matrix	III-40
Figure III-33	Hearsay II representative results	III-42

Figure IV-1	Markov processes	IV-3
Figure IV-2	Improper initial condition	IV-10
Figure IV-3	Loop with immediate-recurrent states	IV-11
Figure IV-4	Incompatible sequence	IV-12
Figure IV-5	Link split paths	IV-13
Figure IV-6	Process split paths	IV-13
Figure IV-7	Process combinations	IV-18
Figure IV-8	Adjacent ports of a process	IV-21
Figure IV-9	Ports attached to SOURCE/SINKS	IV-23
Figure IV-10	Combining processes that are in-parallel	IV-25
Figure IV-11	An irreducible graph	IV-29
Figure V-1	A ring of processes	V-2
Figure C-1	Bliss/11 FIFO 6 processors evaluation data	C-2

Chapter I

Problem Statement, History and Goals

I.A. Introduction

This research develops both a methodology for enhancing the design of programs to be composed of concurrently executable subparts and a set of tools to support that methodology. The execution environment which we shall be concerned with consists of several processing units operating under the control of separate instruction streams. Intuitively, when parts of a program are processed in such an environment, the real time[†] required to execute the program should decrease[‡]. For this reason, as well as others, much current research effort addresses program structure for just such a multiprocessing environment. This thesis addresses the problem of decomposing programs for concurrent execution in such a way that the decompositions are efficient with respect to certain specifiable criteria. The approach is to provide a set of tools with which a system designer can manipulate and analyze a program model created to predict the performance of a system designed for a multiple asynchronous instruction stream environment. The tools are applicable to both the early design of a program and later tuning of a program under construction.

[†]"Real time" is the time elapsed between the start of computation and the time the final result is available. It is different from the total processing time since operations may be performed concurrently.

[‡]This does not always occur. Graham [Graham 72] has shown that adding more processors can increase real time due to scheduling anomalies.

There are several reasons why many researchers are considering multiprocessing and problem restructuring in favor of merely building faster computer hardware without explicit concurrency. First, certain problems overwhelm current and projected technology when programmed for single instruction stream computers. An example is the problem of weather forecasting for any single place on the earth. At present, this problem can not be solved with enough lead time to make the forecast useful. Another large problem is fast-response scheduling, cost accounting, and resource management for large corporations. In this problem the mathematical computations are not necessarily as complex as those for weather forecasting, but the amount of data processing required can be extremely large and, as for weather prediction, there is a time constraint on the answers. For each of these problems, a solution might be attainable in a reasonable period of time if some of the computations could be distributed and executed in parallel. Among the unknown factors are how the problems should be decomposed for distributed processing and what communication constraints and processing attributes elicit favorable computational attributes (such as real time speed and low cost).

There may also be economic incentives to implement a program in a multiprocessing environment. For example, it may be less expensive to implement a speech understanding system on a set of minicomputers than on one fast and relatively complex uniprocessing computer. The price benefits may occur because of

1. the use of so called off-the-shelf equipment making total processing power cheaper than large uniprocessing machines, and
2. economies of scale in manufacturing.

Perhaps the most compelling reason (possibly a consequence of the first two) for wanting to decompose programs for multiprocessing environments is that such

environments are now available and it is important to use them properly. C.mmp [Wulf 72b], BBN Pluribus IMP [Heart 73], the Burroughs D825 [Anderson 62], UC Berkeley's Prime [Quatse 72], and UC Irvine's DCS [Farber 75] all have some multiprocessing capabilities. Additionally Clark's macromodules [Clark 72], Bell's register transfer modules (DEC PDP-16) [Bell 72] and the similarly oriented projects of Fuller and Slewlorek [Fuller 73], and others offer multiprocessing on a very low level.

There are, at present, no guidelines for decomposing a problem for multiprocessing execution [Newell 75]. A number of questions related to the discovery of such guidelines have been investigated. These include.

1. Can a problem be decomposed for solution in a multiprocessing environment? [Karp 66, Gosden 66, Miranker 71, Dennis 71, Anderson 65, Rosenfeld 69]

2. How can the algorithmic structure of a multiprocessing task be represented? [Adams 70, Baer 70, Bredt 70, Dennis 71, Dennis 73a, 73b, Karp 69, Lesser 72, Miller 73, Noe 73]

3. Will the same results always occur, namely will a multiprocessing system be deterministic? Can a multiprocessing system be proven correct? Are there potential deadlocks and unattainable states? This is somewhat analogous to discovering infinite loops and impossible conditions in a sequential program. [Karp 69, Keller 73a, 73b, Riddle 72]

4. When are two computations the same? [Karp 69, Keller 73a, 73b]

5. What measures are interesting about the computation? Some may be: speed, redundancy, (in)efficiency, resource utilization, and economies of the components. [Browne 73, Lehman 66]

6. How can the system be scheduled when there are scarce resources? [Adam 72, Graham 72]

7. How can bottlenecks be identified and their effects lessened or eliminated? [Courtois 72, Dijkstra 74, Rice 73]

8. What are the effects of restructuring the communications among the cooperating processes? [Balzer 71, Horning 73]

9. What style of decomposition and machine structure would best suit a particular programming system (eg. Illiac IV, STARAN, STAR-100, ASC, C.mmp, etc.)? [Flynn 66]

environments are now available and it is important to use them properly. C.mmp [Wulf 72b], BBN Pluribus IMP [Heart 73], the Burroughs D825 [Anderson 62], UC Berkeley's Prime [Quatse 72], and UC Irvine's DCS [Farber 75] all have some multiprocessing capabilities. Additionally Clark's macromodules [Clark 72], Bell's register transfer modules (DEC PDP-16) [Bell 72] and the similarly oriented projects of Fuller and Siewiorek [Fuller 73], and others offer multiprocessing on a very low level.

There are, at present, no guidelines for decomposing a problem for multiprocessing execution [Newell 75]. A number of questions related to the discovery of such guidelines have been investigated. These include:

1. Can a problem be decomposed for solution in a multiprocessing environment? [Karp 66, Gosden 66, Miranker 71, Dennis 71, Anderson 65, Rosenfeld 69]

2. How can the algorithmic structure of a multiprocessing task be represented? [Adams 70, Baer 70, Bredt 70, Dennis 71, Dennis 73a, 73b, Karp 69, Lesser 72, Miller 73, Noe 73]

3. Will the same results always occur, namely will a multiprocessing system be deterministic? Can a multiprocessing system be proven correct? Are there potential deadlocks and unattainable states? This is somewhat analogous to discovering infinite loops and impossible conditions in a sequential program. [Karp 69, Keller 73a, 73b, Riddle 72]

4. When are two computations the same? [Karp 69, Keller 73a, 73b]

5. What measures are interesting about the computation? Some may be: speed, redundancy, (in)efficiency, resource utilization, and economies of the components. [Browne 73, Lehman 66]

6. How can the system be scheduled when there are scarce resources? [Adam 72, Graham 72]

7. How can bottlenecks be identified and their effects lessened or eliminated? [Courtois 72, Dijkstra 74, Rice 73]

8. What are the effects of restructuring the communications among the cooperating processes? [Balzer 71, Horning 73]

9. What style of decomposition and machine structure would best suit a particular programming system (eg. Illiac IV, STARAN, STAR-100, ASC, C.mmp, etc.)? [Flynn 66]

The last question points out that there are several styles of multiprocessing. Flynn [Flynn 66] described processing organization in four ways:

single instruction stream - single data stream(SISD),

single instruction stream - multiple data streams (SIMD),

multiple instruction streams - single data stream (MISD), and

multiple instruction streams - multiple data streams (MIMD).

These computing styles may be used to describe an entire computing environment and affect a problem's decomposition and algorithms. However those systems that do not allow a programmer to program explicitly for multiple streams of data or instructions will be considered as single stream machines. For example, any multiprogramming machine performs some operations concurrently (e.g. I/O), but a programmer is usually unable to control this concurrency. In an array or associative processor a control unit specifies which operation is performed simultaneously on many data items simultaneously -- these are SIMD machines. The current pipeline machines (CDC STAR-100, TI ASC) perform parts of single operations on several pieces of data. The programmer has no control over which operations are performed concurrently, so these are also single instruction stream machines†. Even in multiple instruction stream processing there can still be a spectrum of communication schemes. Networks of computers and multiprocessing computers with common memory are are defined to be multiple instruction stream machines only when a programmer can specify concurrent operations and these operations can be performed concurrently.

A multiple Instruction stream program is defined to be a program in which two subparts of the program can be specified to execute concurrently. Since these are

---

† A pipeline machine has multiple data streams as far as a programmer is concerned, but actually the stream of data comes into the pipe sequentially.

subparts of a total there is some relationship between them. The relationship must be in the form of some common data communication and/or sharing. If the subparts are named A and B then at least one of the following must occur: data progress from A to B, from B to A, from some C to A and B, or from A and B to some C. (Figure I-1 shows the possible relationships between two processes in a directed graph notation) When data progress from one program to another it means that the second program uses some results of the first in its computations. Of course, other processing may manipulate the data between the processing of two subprograms and additional data may be provided to the second program from sources other than the first program (and the first program can provide data to other programs).

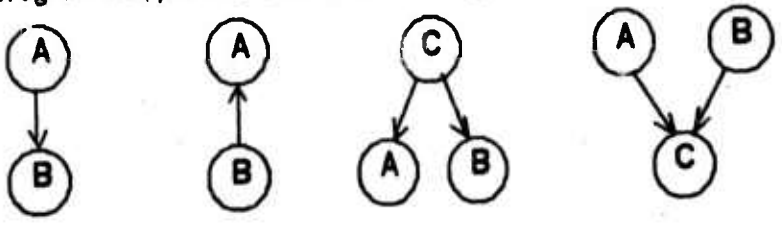


Figure I-1. Possible relationships between two processes, A and B

If A and B are related, one of these relationships must hold; otherwise A and B would be unrelated and thus not subparts of the same program. In the first and second cases one subprogram sends data to the other and continues to process after sending data to the second subprogram. In the third case, data can progress to both A and B from a common source and all three can be processed at the same time. In the last case, A and B can be processed simultaneously and each is able to send data to the same third process, C.

I.B. Direction of this work

At present there are no proven guidelines on how to structure a problem for implementation in a multiple-instruction-stream multiprocessing environment. Rather than address the guidelines problem directly, this work presents a design environment, a set of evaluation tools, and a design approach whereby a system designer can explore attributes of alternative program decompositions. A major premise for this research is that the communication pattern among concurrent processes is critical to a system's performance. The goal is to identify issues and to make predictions which will provide some practical information to the system designer at an early stage and also during later program tuning. This research has been directed towards solving a more specific set of problems than those presented in the previous list, namely:

1. How can interactions among the concurrent computations be modeled?

2. Are the interactions safe, i.e. deadlock free? For example, can one show that a program never arrives at a state in which one process is trying to communicate with a second process while the second is waiting to send a communication to the first process?

3. When the structure is not deadlock free, what is the probability of a deadlock?

4. Where will most of the process and communication activity occur?

5. Where may bottlenecks occur, and how may they be relieved? For example, will the introduction of buffers or additional processes help?

6. Are there working sets of processes? If certain subsets of processes tend to be active at different times then fewer processors will be required for a program (and consequently less parallelism can be attained).

7. What are the effects of restricting the number of processors? What are the effects of alternative scheduling algorithms?

These questions were chosen because they may present hidden problems to

the system designer. Inexpensive and fast approximate answers to these questions should be useful when a program is being designed and also when it is being tuned to improve a program's performance.

Currently there are no generally accepted languages or graphical techniques for representing or modeling a multiprocessing computation and the communication interactions among processes. Thus problems that might be prevented by a clear algorithmic description technique may still occur. However a system designer has some understanding of the relationships among the parts of his system. He can implement the subparts in many different languages, but it is the interfaces between the subparts that are usually not well described. Parnas [Parnas 71] has suggested communication schema to be used while creating communicating modules, but has not described how to represent the communications in an entire system. This lack of global view may prevent the recognition of potential problems. This, then, illustrates the importance of discovering a method for the automatic detection of deadlocked structures and potential deadlocked structures. If the system designer can easily identify in advance where he may have made such an error, then he is spared the task of finding the problem later. It would be preferable to prevent such problems, since many of the criteria for preventing deadlocks are known; however, in complex systems it is increasingly difficult to be aware of all potential deadlock conditions.

If the system designer is able to estimate which particular subparts of his system will contain the largest amount of activity, then these subparts will be the most appropriate places to expend effort to improve performance.

The ability to compare the potential performance of alternate systems easily is extremely important. Almost all disciplines concerned with the creation of large

interacting subsystems use the technique of modeling the behavior of the whole system and extrapolating the performance of this model to deduce properties of the large system. Examples of this technique range from the use of wind tunnels and analog simulation of fluid flow to discrete computer simulations of supermarket check-out counters. A tool for the prediction of computer system decomposition performance should be just as useful. An important aspect of a design system is how easily the designer can alter the attributes of his system and determine the effects of those changes.

We feel that important assets of design tools are that they:

1. be easy to use,
2. provide results quickly,
3. be interactive (when using a computer system), and
4. make it easy to perform design iterations.

I.C. Other work bearing on the problem

Several kinds of tools are available to a system designer. These tools include graph models, queueing theory models, simulation languages, programming languages and theories of design of complex systems. Each of these tools can be useful at some time during the design and construction of a multiprocessing program. Graph models are usually used to represent multiprocessing computations and for analysis of control flow within a program. Queueing theory is used to predict and study performance of simplified models of complex processes. Simulation is an approach to modeling more complex systems to obtain similar performance predictions. Programming languages

are tools for explicitly representing multiprocess algorithms. They also may contain primitive operators that can facilitate proofs of properties of programs. Design theories, such as that of Parnas, provide techniques that facilitate construction of complex systems and their understanding. No one tool is comprehensive enough to use as a quickly obtained predictor of the performance of a multiprocess program.

With sufficient instrumentation the behavior of a multiprocess program can be measured. These data can be used in several ways to predict behavior changes when some system parameters and structures are modified. Again queueing theory and simulation techniques are useful tools for these predictions. As before neither method necessarily provides fast predictions of the sensitivity of performance to changes in program parameter and structure.

The following are brief presentations of some tools that bear a relationship to those that will be presented later. It will be seen that the purely analytic techniques are often too restrictive on assumptions, not useful for overall program design, and of limited applicability due to computational complexity. The simulation techniques require too much effort both to construct a simulation and to modify it to achieve results concerning alternate program decompositions.

I.C.1. Petri nets

After the original formulation of Petri nets [Petri 62] several MIT researchers [Dennis 70, Holt 70, Paterson 70, Rodriguez 67] refined forms of the original model as useful tools for studying concurrent processes. A Petri net looks like a directed graph in which marks or tokens are placed on some of the arcs. (Only connected graphs are of interest.) These tokens move about the graph to represent flow of control. When

tokens are present on all of the input arcs to a node, that node is able to "fire." After a node fires, one token is removed from each input arc and a token is placed on each output arc of that node. In fact, a Petri net is not a directed graph [Berge 62] because it is possible for one arc to point to or come from more than one node. A restricted Petri net called a marked graph [Holt 70] permits arc initiation and termination only at single nodes (not necessarily the same). Multiple arcs can still be connected to each node. In contrast, a restricted Petri net becomes a finite state automaton (state transition diagram [Holt 70]) by only permiting one arc to enter each node and one arc to leave each node. (Arcs can have multiple starting points and terminal points.) In Figures I-2, I-3, and I-4 the nodes are represented by straight lines and the arcs are arrows with a circle that can contain the tokens (represented by dots).

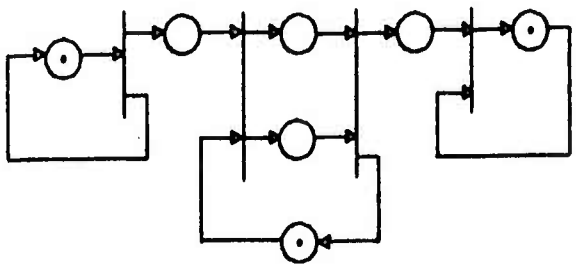


Figure I-2. A Marked graph

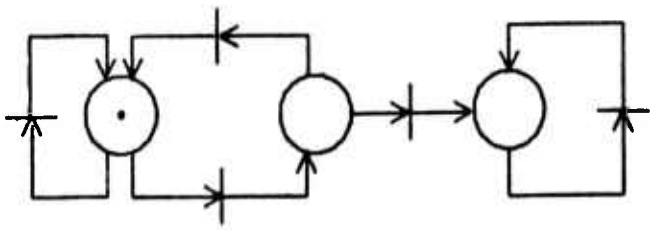


Figure I-3. A finite state atomaton.

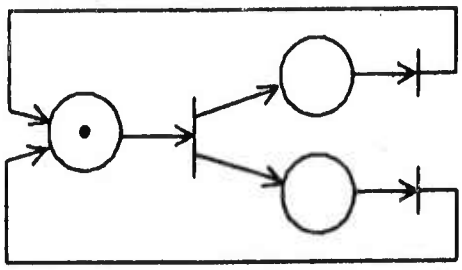


Figure I-4. A Petri net that is neither a marked graph nor a finite state atomaton.

Marked graphs are the only form of Petri nets that have been used to study concurrent processes. The general Petri net can be too complex and the state transition diagram can not be used to model concurrent processing. Marked graphs are used by modeling the potential flow of control in a system and then analyzing possible markings in order to make predictions about future markings. Issues investigated, for a particular initial marking, include:

1. Determine whether nodes will eventually activate (fire). In Petri net terminology the question is whether a node is "safe" [Holt 70]. If all nodes are safe the net is "safe," i.e. all nodes can be activated.

2. Count the number of activations of a node. The important counts are 0, n, and infinity.

3. Determine whether the initial marking can lead to another particular marking.

4. Identify nodes that can fire concurrently.

There are several difficulties in using Petri nets. One is that interesting examples require a large number of nodes [Dennis 70, Merlin 75]. There are so many nodes that it is difficult to do any analysis. In addition, none of the analysis is mechanical. Another difficulty is that control flow in the graphs is completely determined with no accounting for rates of processing at each node.

I.C.2. The UCLA model

The original goal of the UCLA model was to "represent programs to be run on variable structure computers" [Baer 73, Estrin 63]. Thus its purpose was to help describe concurrent computations rather than to study the performance of algorithms. However some extensions of, and associated restrictions on, the original model allow for performance predictions in some restricted cases to determine the termination of loops, the determinacy of representations [Regis 72], and the reduction of graphical forms [Bovet 69]. In addition the UCLA model has been used to study the automatic conversion of FORTRAN-like programs to a parallel computation form.

The basic form of the model is a directed binary graph. Most studies using this model use an acyclic structure. The graph shows processing dependencies and, as long as an acyclic model is used, potentially concurrent operations can be easily identified. Each node may have at most two entry arcs and also at most two exit arcs. The rules for firing a node are defined as part of the node. The node's firing rule depends on the enabling of the input arcs and the node's result rule cause some of its output arcs to be enabled. A node will not fire if any of its output arcs are already enabled. Once a node fires the input arcs causing that node to fire are disabled. (See Figure I-5)

In the UCLA model, branching and merging control flow are modeled with EOR type nodes. Concurrency is modeled by the use of AND type nodes.

Further restrictions are placed on the form of the UCLA graph model. There must be a unique initial vertex (only output arcs) and a unique terminal vertex (only input arcs). Another restriction is that all subgraphs must be AND type. This means that if a choice is made at an EOR output node then it must still be possible for the

AND input node

AND input type fires only if both input arcs have been enabled.

EOR input node

EOR input type fires only if exactly one of the input arcs has been enabled.

AND output node

AND output type enables both of the node's output arcs after the node has fired.

EOR output node

EOR output type enables exactly one of the node's output arcs after a node has fired (which one is undetermined).

Figure I-5. UCLA Model Nodes.

terminal node to fire. In addition it should not be possible for both arcs of an EOR input node to be enabled at once. The question of determinacy of a graph is subsumed by the question of legal graphs. Legal graphs are those that start at the initial node and are guaranteed to terminate at the terminal node. When loops are

allowed, any loop must be able to terminate. Most who have used the model have assumed acyclic structures in order to guarantee loop termination (naturally, no loops). The analytic technique used to ignore loops is to expand all loops by some finite repetition. The repetition factor is determined by a probabilistic argument [Martin 67].

The question of mean path length in a directed acyclic binary graph has been studied at UCLA. Probabilities are assigned to each arc and computation times are assigned to each node. These are used to determine the probability of traversing paths through a legal graph and to estimate the mean path time of a graph [Martin 69]. One may also determine the maximum number of processors required by the graph [Baer 69] under the same restrictions.

The difficulties with using the UCLA model also involve the need for a large number of nodes to represent interesting structures. This is particularly true since each node has at most two input arcs and at most two output arcs. Another problem is that most results have been dependent on acyclic models. Thus the mechanical techniques for proving legal graphs, etc. are only applicable to a restricted set of programs representable by the model.

I.C.3. An algebraic model of interprocess communication

In his dissertation [Riddle 72], Riddle presented a methodology for modeling and analyzing supervisory systems, but the work can be applied to the problem of analyzing any complex asynchronous system. He found the same difficulties with Petri nets and other models as those reported in earlier sections of this chapter.

Riddle presented an explicit program-like description of the operation of a process. This description was only concerned with the interprocess communication

relationships of each process. However the description was close enough to being a program that each process required information concerning the type of interprocess messages. Therefore the descriptions of processes were themselves fairly complex.

The model was also based on a directed graph structure representing interprocess communication. A graph consisted of two types of nodes, process nodes and link nodes. The link nodes had properties that could require a certain amount of computation associated with them, e.g. queueing disciplines.

One of the goals of Riddle's research was the development of an algebra to describe interprocess communication. Algebraic expressions could be used to describe possible communication paths in a model. By using the graphical structure, the program-like descriptions, and the algebraic expressions, theorems were developed to analyze the behavior of a modeled system. The creation of all algebraic expressions is performed by the inspection of a graph. The proof of theorems concerning the behavior of a system, as described by the algebra, is not a mechanical process. Riddle did provide a set of theorems that can be used in a proof.

The examples that Riddle studied were based on communication paths of a given system. He determined what termination and deadlock meant for that system and was able to derive proofs showing that the system terminated and contained no deadlocks. The questions he posed were specific to the system being modeled and required the creation of algebraic expressions for each question concerning system behavior. These algebraic expressions were not necessarily easy to create and the proofs of theorems were not very easy to construct.

The tools that Riddle's research provides may be used for the design of multiprocessing programs. The drawbacks to his approach are the difficulty and effort

required to create the algebraic expressions needed to represent a model, and the expressions representing communication within a model. The expression proof process is also fairly tedious.

I.D. The STEPPS System

All of the models discussed in the previous section are tools for the analysis of multiprocess programs. A common drawback of each model is that results must be obtained through detailed, non-mechanical analysis. A second drawback is that none contains the processing rates of the various processes of a multiprocess program as part of the model. The speed and ease of obtaining results and the ability to include expected timing of attributes of a program can be especially useful when making early design decisions concerning the structure of a program.

The design methodology presented in this thesis is based on an interactive system utilizing a particular model of muitiple instruction stream problem decomposition. The system and the model are called STEPPS (Some Tools for Evaluating Parallel Processing Systems). The methodology of designing a programming system has become an interesting and important question in the last few years [Brinch Hansen 74, Dahl 72, Mills 71, Parnas 72, Parnas 75, Weinberg 71]. The author subscribes to the "top down" approach to system design [Simon 62]. Thus a "natural" approach to building a system that will contain potentially concurrently executing subparts is to decompose a system into functionally independent subparts and describe the communication structure among the subparts before explicitly defining the operation of the subparts. For example, when designing a compiler one might say that

the LEXICAL-ANALYSER and the SYNTAX-ANALYSER could process in a pipeline manner with the LEXICAL-ANALYSER sending results to the SYNTAX-ANALYSER. A convenient notation is a directed graph notation with the restriction that the connections between two processes must go through an explicitly designated (and named) connecting LINK (see Figure I-6). At this stage of decomposition only potential communication is important and data dependent communication (i.e. decisions based upon data) is not considered at all.

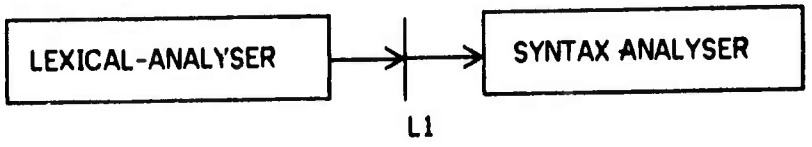


Figure I-6. Pipeline.

Each of the previously discussed models considers interprocess communication patterns to be important for understanding the performance of multiprocess programs. Both the Petri-net and the UCLA model represent interprocess communication by means of the movement of untyped tokens. Queueing models of multiple processes also use typeless tokens to represent flow of control. Riddle was able to simplify some interprocess connections in his model schema and to enhance analysis by introducing type identification for tokens.

The STEPPS model uses typeless tokens to represent flow among processes. The study of interprocess communication suggests several measures of multiprocess performance such as queue lengths, deadlocking, and potential concurrency. A difference between this model and the earlier models is that a STEPPS process must be ready for a message before "firing" (due to the arrival of a message) instead of its

firing being dependent on logical relations of the messages available on paths to it.†

I.D.1. The STEPPS model: an informal description

The STEPPS model includes both probabilistic and timing expectations for describing individual process activity. Whereas a standard probabilistic model, i.e. Poisson, treats processes as operating on messages, the STEPPS model process includes a natural relationship between a process' input/output activities. In addition, the introduction of time parameters allows for better estimation of a program's operational concurrency instead of potential concurrency. The model represents multiprocessing at the message communication level and is not intended to represent other multiprocessing problems such as memory interference and specific programming techniques.

Concurrency can be modeled by having a single process send data to more than one other process. Data streams are explicitly merged when a process receives data from more than one other process. In the descriptions of processes, more than one arrow may leave a process node or enter a process node. If a process node is able to receive data from any of several processes, but the receiving process does not care which process sent the data, several arrows enter a linknode and only one leaves it. If one of several processes may operate on data produced by another process this is represented by more than one arrow emanating from a linknode and going to the separate process nodes.

---

†In Riddle's model a process must be explicitly programmed to accept a message.

Example I.D-1

> Consider the problem of building an online university registration system. This system would handle all of the scheduling and student record keeping for a university. One might decompose the problem into the graph of Figure I-7. Students' requests are handled either by a Schedule Requester or Schedule Updater, each of which processes the request and sends data to a Scheduler. The Scheduler sends data to the Data Base and then sends results to a Schedule Output process. Requests to the system may also come from the Registrar. These requests may also go to the Data Base and on return data is sent to the Registrar Output by the Transcript request process. There also may be requests for grades. The data base may access data in either the Current Semester or its Archives.

All data travels through paths between nodes (the ***LINKS*** and the ***PROCESSES***) in units called messages and all queueing of messages occurs at each LINK. Requests for data from a LINK are handled in a FIFO (First In, First Out) manner by the LINK. The next step is to describe the action of a PROCESS node. Since only the communication paths are important at this point of design, only the message handling properties of a process are described. The source of data is not identifiable, so a process neither knows which process sent the data to the LINK attached to any of its "input ports" nor does it know which processes are attached to the LINK that is attached to any of its "output ports." The reason for this restriction is that messages contain no information such as sender or receiver identity. This will be shown not to cause difficulty in using the model.

The execution sequence of a process is:

1. perform an input or output operation,
2. choose which input or output port will be active next,
3. compute for some time, and
4. repeat 1 to 4.

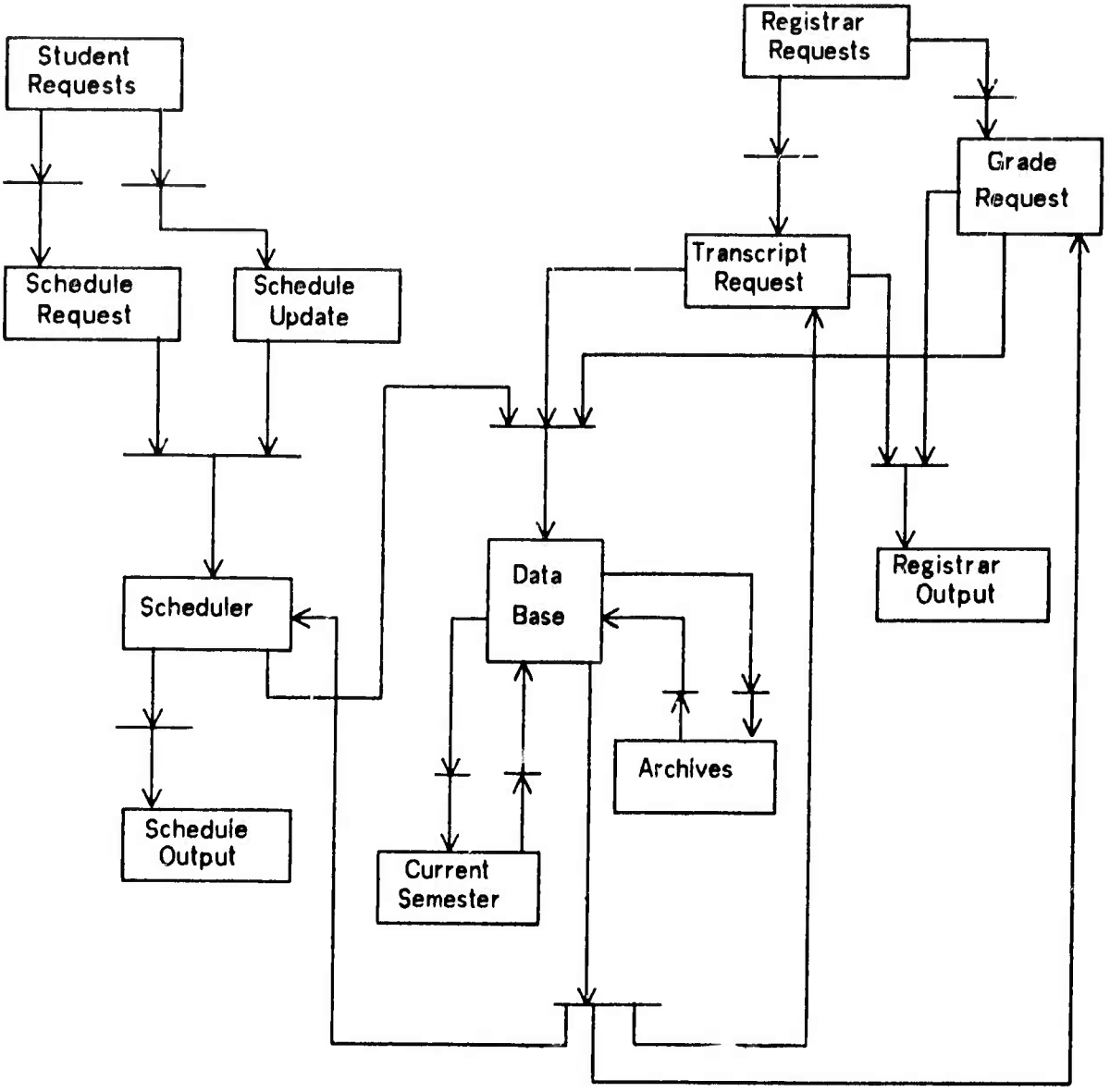


Figure I-7. Registrar's Data Retrieval System

Each process is a uniprocess and can only perform one input or output operation at a time.

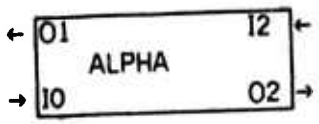
The method of describing how each process operates in the STEPPS model requires that each port be named. For convenience, the notation used is to assign a

type of either "I" for input or "O" for output and a number. A transition matrix for each process defines the probability of succeeding the activation of one port with the activation of another port. The informal definition *state of a process* refers to the most recent port activation (in this thesis, *states* correspond to port activations). The process remains in the same *state* while it is computing and enters a new state at the next activation of a port. In most contexts the terms "state" and "port" are used interchangeably.

Example I.D-2

ALPHA is a process with two input ports, I0 and I2, and two output ports, O1 and O2. I0 may transfer to state O1 or O2. I2 may transfer to state O1 or O2. O1 may only transfer to state I0. O2 may transfer only to state I2. The graph and transition matrix for this process is shown in Figure I-8.

Graph notation



Transition matrix (without timing)

ALPHA	I0	I2	O1	O2
I0	0	0	a	1-a
I2	0	0	b	1-b
O1	1	0	0	0
O2	0	1	0	0

Figure I-8. Process ALPHA

The transition matrix makes it possible to describe the splitting of processing, the merging of processing, and the choice of alternate computation paths. In Example I.D-2, after an input from port I0, process ALPHA can enter either state O1 or O2 (with

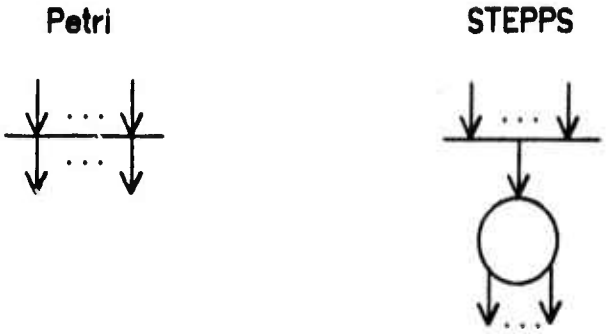
probability of "a" to O1 and "1 - a" to O2). State O1 always enters state I0 as the next state.

Other features of a STEPPS model that can be specified are:

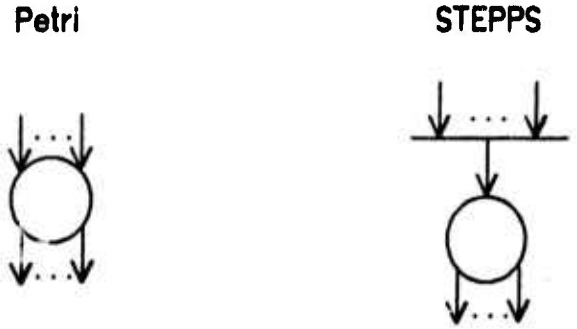
1. the initial state of each process,
2. the number of messages that a port may receive or send before the process changes state,
3. the amount of computation time, defined for each transition, that a process computes before a transition takes place (this is fixed, but random variable computation times can be approximated),
4. the amount of computation time taken by a LINK to accept or send a message, or to restart when it is not already handling communication of messages, and
5. the queue size limits for each link and the initial number of messages in each link.

The model that has just been described subsumes both the Petri net and the UCLA model. The links and nodes of Petri net and STEPPS models are very similar; each is equivalent to the corresponding STEPPS model shown in Figure I-9. Figure I-10 also shows the relationships between the UCLA model and the STEPPS model.

The STEPPS model allows for a more general specification of data flow than the earlier models since it is possible to describe the probabilities that particular data paths may be taken. When a message is accepted by a process it is easy to specify which data paths are more likely. As further information about the system being designed is learned or when the effects of alternate data path specification are taken into account, probabilities are altered by the system designers to fit the new structure.



The input port accepts N messages before changing state to an output port; the transition between the output ports occurs in a sequence; and the last output state transfers to the input port.

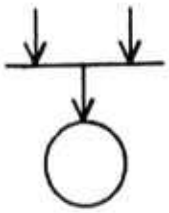


The input state accepts only 1 message and the transition to each output state is equally likely. Each output state transfers to the input state. The link may be able to hold more than one message.

Figure I-9. Mapping Between Petri nets and STEPPS model.

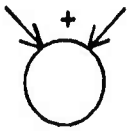
UCLA

STEPPS

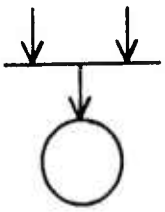


The input state accepts 2 messages before transferring to an output state.

UCLA



STEPPS



The link has a limit of one message, so only one message can get to the process. The process input port accepts 1 message before transferring to an output state.

UCLA

STEPPS

The transition matrix sequences through the two output ports.

UCLA

STEPPS

The transition matrix shows an equal likelihood of transferring to each output port from the input ports. After an output the process will perform an input.

Figure I-10. Mapping of UCLA model to STEPPS.

I.E. The STEPPS system and simulator

Once a proposed multiprocessing program has been modeled, the model can be implemented in the STEPPS interactive system in order to evaluate the particular decomposition. The data entry language for STEPPS has been designed for conciseness. A linear description of a directed graph and the associated transition matrices may require the entry of a fairly large amount of data. To facilitate the entry of these data, it is possible to recall previously stored data. The system designer can manipulate his model in any way he chooses, e. g. remove nodes, change parameter specifications, or display parts or all of his model. It is always possible to save the description of the model or parts of it externally in a form that may be recalled by the STEPPS system or examined on hardcopy.

Several useful tools are available to help the system designer evaluate the structure of his decomposition. As a basic step, a STEPPS model can be certified as being a ***well-formed model***. A STEPPS model is well-formed when:

1. For each process, every state is attainable from any other state (If a process has N states, and X and Y are any two of them (possibly the same), the probability of starting in state X and entering state Y in N or fewer transitions is greater than zero. This restriction is discussed in later chapters.); and

2. All ports of each process are attached to links;

3. All links are attached to both input and output ports;

4. The graph is connected. (When the directions of paths are ignored then there exists a path between every pair of nodes.)

At some point it should be possible to simulate the execution of the modeled program; thus the STEPPS system contains a model simulator. However, there remain problems which can prevent a successful simulation of a program structure. One

problem is that the initial state of the processes and the initial message capacities of the links might be incompatible. This would cause a simulation to halt almost immediately. Another problem is one of possible communication deadlocks. These problems are discussed in Chapter IV.

I.E.1. Deadlocks

A process may deadlock in either of two situations:

1. no messages will ever be available at the link attached to an active input port, or
2. the capacity of the link attached to an active output port has been reached, and no messages will ever be able to leave the link.

Deadlocks may occur when a process can depend on itself improperly. They may also occur when a set of processes are incompatible for reasons other than data loops.

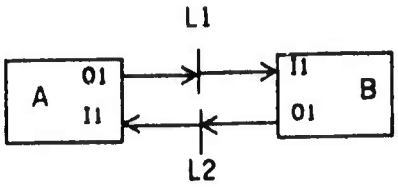
Example I.E-1

Figure I-11 shows process A waiting for data from B while B is waiting for data from A. If the initial state of A is changed to be O1 then the process has no deadlocks. If an additional change is made to A so that state O1 or I1 is activated more than once and B is not changed, then this is again unsafe because a link will eventually overflow or never have enough data.

Example I.E-2

Figure I-12 shows a non-loop structure where there will be a deadlock as soon as the L1 queue limit is reached.

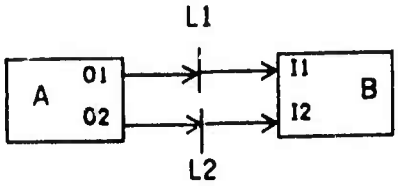
None of the structures presented in Examples I.E-1 and I.E-2 showed problems that can occur when a process has a choice of successive states. There are other



A	I1	O1
I1*	0	1
O1	1	0

B	I1	O1
I1*	0	1
O1	1	0

Figure I-11. Incompatible loop.†



A	O1	O2
O1[2]*	0	1
O2	1	0

B	I1	I2
I1*	0	1
I2[2]	1	0

Figure I-12. Incompatible non-loop.‡

deadlock producing structures. For example a process may be set up to produce either N or M messages and the safety of this structure must be recognizable.

---

†The asterisk in the example means that this is the initial state.

‡O1[2] means two occurrences of O1 before changing state.

The deadlock problem may be dealt with in two ways. One alternative is to require any program decomposition to be deadlock-free.

A second alternative is to determine where deadlocks may occur and the probability of a deadlock. The existence of deadlocks in real systems is not always bad as long as a suitable response can be made. For example, the ARPA network is not deadlock free [Kleinrock 75]. However, when a deadlock is suspected the system "times out" and requires reinitialization of a data message. This is a reasonable solution under some circumstances, but only when a system can lose information. STEPPS provides tools to recognize the possible occurrence of deadlocks.

The algorithms used to identify deadlocks are basically specialized graph reduction techniques. A model is viewed as a graph whose nodes are the processes and links. Under application of these reductions, a safe model will collapse to a graph containing no nodes. If the graph does not collapse then a deadlock is possible.†

As already noted, a STEPPS model of a program can be quite general. If a model is acyclic and meets other criteria set by Martin [Martin 69] it is possible to estimate mean path time; however, these criteria are quite restrictive. In general it is not possible to estimate mean path time through a STEPPS model without simulation. Ordinary systems analysis techniques such as queueing theory and dynamic programming models are intractable in all but the simplest cases [Fishman 73, Gordon 69]. It is for this reason that a simulator is a basic part of the STEPPS system.

The simulator is easy to use since it is a specialized system and requires no programming. Naturally, any STEPPS model can be simulated using GPSS, SIMSCRIPT, SIMULA, or any other simulation language. However the effort required to reprogram

---

†Chapter IV discusses the deadlock problem and past work in the area.

a general model is not well spent at the design stage of building a multiprocessing program. It is at this early stage of program development that the designer needs information about possible program decompositions, and the flexibility to be able to alter his design easily and make new evaluations. If variations in the decomposition needed to be reprogrammed, the understanding of alternative systems would be a more difficult process than comparing alternative models using the STEPPS system.

A variety of simulation parameters can be easily altered for comparing their effects on simulation results. These parameters include: restricting the number of available processors, identifying processor competing and noncompeting processes, and varying process scheduling algorithms.

The STEPPS simulator has a set of data gathering functions which help the designer evaluate a particular decomposition. Some of the estimations that are made based on the data are:

1. The expected time that each process is in each state. This can be determined without simulation if only the processing time is of interest, but when process wait time is included it is too difficult to estimate the time spent in each state.
2. The expected number of messages in each queue.
3. The expected number of processes waiting to send a message to each link.
4. The expected number of processes waiting for a message from each link.
5. The expected number of processes that will be executing simultaneously. This can be used to estimate the number of processors needed.

This list is not complete for all uses of the simulator. The system has been designed so that it is not difficult to include additional measurement functions.

The simulation times required to obtain these estimates vary with the complexity of a model. Usually, useful estimates can be obtained with a few minutes of

DEC PDP-10 compute time. The complexity of a graph is dependent on such attributes as the number of connections, choice of process states, and link delays.

I.F. Thesis contributions and outline of remainder of thesis

The contributions of this thesis are tools that a system designer can use to enhance the overall design of a multiprocess program. These tools, presented as the STEPPS system, are based on a model that is described precisely in Chapter II. Chapter II also discusses the STEPPS system's capabilities (Appendix A is a manual for the STEPPS system). Examples of how the STEPPS model can be used to model a variety of multiprocess structures are presented in Chapter III. In addition, Chapter III presents two larger examples: one of the use of the STEPPS system in a user's early design stage and the other of the use of the STEPPS system in system tuning. The deadlock reduction algorithm is presented as a set of theorems with proofs in Chapter IV. Other model analysis capabilities are also discussed. The STEPPS simulator and data gathering facilities are discussed in Chapter V. Chapter VI contains a review of the thesis results, the limitations of this research, conclusions and directions for further research.

Chapter II

The STEPPS Model

This chapter provides some formalisms for later use and a precise definition of the STEPPS model. Chapter I presented an informal description of the model and the interactive design environment based on the model.

II.A. Modeling the behavior of a process

The term *process* describes the utilization of the processing unit of a single instruction stream-single data stream computer (SISD). A "process" has sometimes been defined as the execution of a program. For the purposes of this research, that definition is too limited, since it does not take into account data transfers and accesses. A process, as defined for the STEPPS model, exists in one of the following conditions:

1. processing (computing) before performing an input or output operation,
2. waiting to access an external resource that must be accessed exclusively (simultaneous accesses are modeled by allowing zero time between accesses), and
3. waiting to complete an input or output operation.

A process is modeled as a processing unit which can perform operations internally, and which then must communicate with other units through one of several *ports*. The communication occurs when a process either requests or provides a unit of information. Each port belonging to a process has only one function: Input to the process or output from the process.

The internal operations of a process are unknown to an observer of a process.

All that can be determined is the relationships among the activities of the process's ports. Externally these relationships appear as probabilistic transfers of activity from one port to another, plus a computation time between port activities. In general, the computation time between any two successive port activations is dependent on the particular ports. Such process activities as accessing resources and sharing resources are modeled in terms of interprocess connections and message flow.

As defined in Chapter I, the *state of a process* refers to the most recent activation of or attempt to activate a port. This was an informal use of the term "state" since, more precisely, a process can be in the state of waiting to activate a port, activating a port (doing the port's activation), computing before the next port activation, etc. The imprecise definition of "state" will be used in most contexts, and it will be made clear when the more precise meaning is used.

This definition of the state of a process is an abstraction based on potential communications between a process and other processes. In addition, the concept of time is included in the model to allow a designer to include processing time for computation during simulations. An important abstraction is that STEPPS processes are not deterministic since port activations are based on probabilities and not on a data directed control structure. The disadvantage in this is the inability to represent programs on an instruction level. The advantage is that all potential communication alternatives are emphasized.

The complete operation of a process is described by the following loop (assuming the process starts in some initial state):

1. Perform the input or output operation associated with the present state. This may involve waiting to access an external resource and waiting for the input/output operation to complete. Both waiting times are considered as time spent in a state while not processing. This step can be repeated a specified number of times before the next step.

2. Choose a new state. By a probabilistic method, described below, a successor state is chosen, but not yet entered.

3. Process (compute) for a length of time as determined by the transition from the present state to the next.

4. Enter the new state and repeat 1 through 4.

Given the knowledge of the present state, probabilities for entering any of the process's states are defined. Since the state of a process is related to the activity of a port, probabilities are defined for potential successive port activations from every port activation. Note that the choice of a successor state is dependent on the present state. In addition, step 3 above implies that a processing time parameter is associated with each transition and step 1 suggests that possible communication time is associated with a port activation.

Two restrictive assumptions are basic to this model. They are that (1) a process can not be interrupted (i.e. the transition matrix completely describes a process' activity) and (2) processes are neither created nor destroyed dynamically. These restrictions are used to keep the model relatively simple; they also make it possible to perform the deadlock test by graph reduction (Chapter IV).† The lack of dynamic process creation and destruction can be approximated by including multiple copies of processes, but there is no way to use the STEPPS system to model process interrupts and preemption.

II.B. Data flow and links

The previous section refers to units of information that are either requested or produced by a process. A unit of information is called a *message*. The number of

---

†These restrictions are examples of a tradeoff between analysis and representations. Some system structures might have been easier to represent if there were, for example, typed messages or dynamic process creation. However, automated system structural analysis was not found to be feasible when these richer representations were considered.

messages in a STEPPS model need not be conserved. Thus a process may successively request messages from each of two input ports, yield a single message on an output port, and then request more messages from an input port. A property of a message is that it is only a token of information. It does not actually contain any information used within the model. A process can not use the contents or type of a message to decide on future activity. Only the existence of a message is meaningful to a STEPPS process. This restriction will be shown not to affect substantially the class of program structures that can be modeled with the STEPPS model. The major restriction is that processes are completely defined by their transition matrices and can not be preempted. Thus systems that contain parent/sibling process dependencies where the parent process can stop, restart, or terminate a sibling process can not be modeled.

Processes are connected via ***links***. Each port of every process is connected to exactly one link, but a link may be attached to several ports of both input and output variety. Messages enter a link from output ports and leave a link going to input ports. Requests for messages from input ports are handled on a first in - first out basis.

The link is the resource that can only be accessed by one process at a time. This access may take zero time, but the restriction is used to prevent race conditions. For this reason the STEPPS model includes a method that guarantees mutual, exclusive access to a link. Since a process may only perform one input or output operation at a time, it can only access one link at a time, so there is no opportunity for a "deadly embrace"† due to the accessing of links.

The STEPPS model can be used to model the situation where there is a non-zero overhead for message transmission. The properties of a link are:

---

†A deadly embrace, as defined by Dijkstra, Habermann and others, occurs when two processing objects are able to reserve more than one resource at a time without all resources being reserved initially. For example; process A reserves resource x; process B reserves resource y; process A needs resource y and can not continue until B relinquishes it; process B needs resource x and can not continue until A relinquishes it. Neither process A nor B will be able to continue.

1. It can store a limited number of messages.

2. It may take a certain amount of *delay time* to either accept or transmit a message (same delay time for accepting or transmitting).

3. Time may be required to start up a link when it is not already active[†].

4. It may initially contain a specifiable number of message tokens.

5. It can receive requests for messages and transmit a message to a requestor if a message is available or force the requestor to wait in a queue (whose size is dependent on the number of processes attached to the link) until a message becomes available.

A link is not a process but its operation can cause timing delays. When a link has a start-up time parameter set to be greater than zero then the link's *start-up time* is significant. The other reasons that a link can force a process to wait in a state are:

1. The link that is attached to the current state's port is already in use.

2. The link has reached its limit of messages and the current state's port is an output port.

3. The link has no messages and the current state's port is an input port.

4. The link's defined delay time is taken to perform an input/output operation.

The complete operation of a link in the STEPPS model is described by the following loop:

1. Do nothing until a process requests the use of the link. Wait for a specific start-up time (if any).

2. If the request is for the link to accept a message and if the link's specified message limit has not been reached, then accept the message. Otherwise do nothing, forcing the process trying to send a message to wait until a message is removed from the queue.

3. If the request is for the link to provide a message and if there are any messages available, then send a message to the process requesting a message. Otherwise do nothing, forcing the process requesting a message to wait until a message is sent to the link.

4. Wait a specified amount of time (if any) for data to transfer.

[†]A similar situation occurs in a virtual memory system when extra time is necessary to bring a page that is not currently in use into main memory.

5. Allow the process that is currently accessing the link to continue.

6. If a process is waiting for a message or waiting to send a message then repeat 2 to 6 (the queue discipline is FIFO). Otherwise repeat 1 to 6.

II.C. Notation and definitions

The notation that will be used in the remainder of this thesis is described here. Wherever possible the linear notation will be the same as that used as the command language and display language for the STEPPS system. (See Appendix A for complete definitions and explanations.)

The attributes associated with a process are: its ports, the links attached to the ports, its transition matrix, its initial state, and the number of repetitions of each state that occurs before the process chooses a new state. The attributes of a link are: the ports attached to it, its queue size limit, initial number of messages in its queue, time to accept or send a message (delay time), and the time to restart a link that has been waiting for activity.

II.C.1. Notation

The following informal and incomplete BNF defines some† of the syntax of the STEPPS system used to describe the attributes of the process and link nodes. The usual definitions for letter, number, digit, and other non-terminals with common descriptive names are assumed.

Generally used terms:

```
<name>          ::=  <letter> | <name> <digit> | <name> <letter>
<process name>  ::=  <name>
```

---

†Some examples will use syntax not shown, such as: everything to the right of "!" is ignored and the words Attribute, Queue, Volume, etc. can be abbreviated. The complete syntax is defined in Appendix A.

<link name> ::= <name>
<port name> ::= <process name>.<port type><up to 3 digits>
<port type> ::= I | O
<input port> ::= <process name>.I<up to 3 digits>
<output port> ::= <process name>.O<up to 3 digits>

Connections between ports (simple connections):

<connection> ::= <input port>←<link name> | <link name>←<output port>

Transition matrices:

<transition definition> ::= <port name><repetition factor> = <initial state><transition probabilities and times>
<repetition factor>::= <null> | [<positive integer less than 262144>]
<initial state> ::= <null> | *
<transition probabilities and times> ::= <port id, prob., compute time> | <port id, prob., compute time>;<transition probabilities and times>
<port id, prob., compute time> ::= <port type> <up to 3 digits>: <prob comp>
<prob comp> ::= <probability>|<probability>,<compute time>
<probability> ::= <real number between 0 and 1>
<compute time> ::= <a non-negative real number>|<null>

Link attributes:

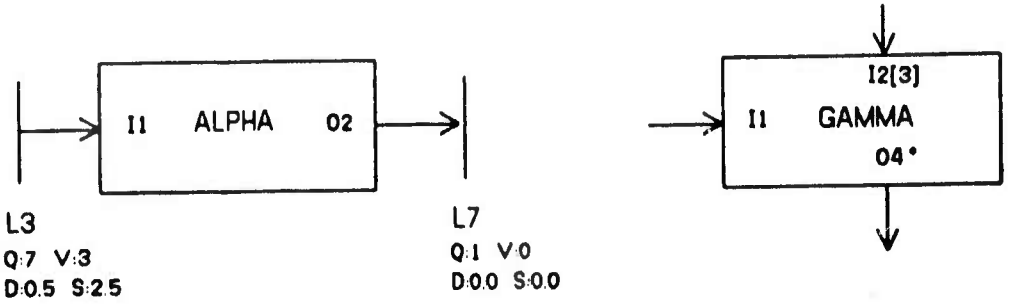
<link attributes> ::= Attributes <link name> <list of attribute definitions>
<list of attribute definitions> ::= <attribute definition> | <attribute definition>, <list of attribute definitions>
<attribute definition> ::= <attribute name> : <number>
<attribute name> ::= Queue | Volume | Delay | Startup

Example II.C-1

ALPHA, L3, L7, and GAMMA are legal process and/or link names. Consider the following STEPPS commands:

```
ALPHA.I1 ← L3
L7 ← ALPHA.O2
Attributes L3 Queue:7, Volume:3, Delay:0.5, Startup:2.5
GAMMA.I2[ 3 ]= I1: .4, 3.5; I2: 0; O4: .6, 1.5
GAMMA.O4 =* I2: .5; O4: 0.5, 7.5
```

The first two lines are examples of the notation for connections. The third line displays the attributes of a link. The last two lines show how transition probabilities are represented. Thus the probability of entering GAMMA.O4 from GAMMA.I2 is 0.6 and will take 1.5 units of time. All STEPPS displays will order the ports of a process in numerical order with input ports before output ports. In addition, missing parameters are defaulted (e.g., GAMMA.I1 probabilities).



GAMMA	I1	I2	O4
I1	0.333,0.0	0.333,0.0	0.334,0.0
I2[3]	0.400,3.5	0.000,0.0	0.600,1.5
O4*	0.000,0.0	0.500,0.0	0.500,7.5

Figure II-1. Process and link graphical notation†

Only one port of a process will have a * when a process's entire transition matrix is displayed.

A graphic notation used in later sections and chapters is shown in Figure II-1 of the last example. A process is a convex figure and will be represented by either boxes or circles. Links will always be represented by straight lines. Connections will be represented by lines with arrow heads denoting the direction that a message would flow.

Chapter III contains an example set of simple and complex STEPPS models of program communication structures. These examples demonstrate that the STEPPS model is expressive enough to represent both toy and non-toy structures while eliminating the details required by programming languages and the details required by a Petri-net like model.

---

†Not all of the attributes of a process or a link will be displayed in later examples.

II.C.2. Summary of parameters to the STEPPS model

The following is a complete list of the parameters that must be supplied for a STEPPS model:†

1. A connection between each process port and a link. Default port connections are to link "DANGLING".

2. A transition matrix for each process showing the probability of entering a new state from each state and the amount of processing time taken before the transition. Default probability values are determined by assigning equal parts of any unassigned probability to each defaulted transition. Default compute times are zero.

3. The initial state of each process (O4*). The first port defined is the defaulted initial state.

4. The number of times a port activation can repeat before a new state is entered (I2[3]). Default is 1.

5. The maximum queue length allowed for each link (Q:7). Default is 1.

6. The initial number of messages in each queue (V:3). Default is 0.

7. The delay time caused by the operation of each link (D:2.5). Default is 0.0.

8. The start-up time to wait when using a link not already in use (S:0.5). Default is 0.0.

II.C.3. Graph definitions

The STEPPS model is a graphical model and thus some abstractions have proven useful in discussing the model. When a graphical structure is similar to that of classical graph theoretical abstractions, the classical structure name has been used‡. Some useful definitions are:

Node -- A node is either a process node or a link node.

---

†The STEPPS system assumes default values for some of the parameters. Examples refer to Figure II-1.

‡Many of the abstractions are based on the text by Berge [Berge 62].

Path -- A path between two nodes is a sequence of nodes with each node connected to the next one in such a way that a process is connected by an output port to a link that is connected to an input port of the next process in the path. There may be many paths between any pair of nodes. A path may include alternate branches as long as each branch leads to the final node of the path.

Adjacent -- An input port and an output port are adjacent if they are both connected to the same link and no other ports are connected to that link.

Attached to -- A port is said to be attached to a link if it is connected directly to the link. An input or output port is attached to a (possibly different) process node if the port is attached to a link that is attached to the process by a link of the opposite type. In particular, there must be a path between the port and the process through only one link.

II.C.4. State definitions

The structure of a process is described by potential transitions among the states. The following abstractions are used when discussing the properties of states of a process:

In-sequence -- A subset of the states of a process is said to be in-sequence if the transition matrix of the process shows that once the process enters one state of the set then, with probability 1, the process will enter the other states of the set in a particular sequential order. In addition, no other state of the process may transfer to any of the elements in the sequence other than the first state.

Onto -- State x is onto state y if for any sequence of transitions starting at x and terminating at the first occurrence of y, state x is not reentered.

One-to-one -- is a relationship between two states of a process occuring when the only way for a state to recur is to enter the other state exactly once and vice versa. State x is one-to-one with state y if x is onto y and y is onto x.

Immediate-recurrent -- A state is immediate-recurrent if it can return to itself in one transition. The process may return to the immediate-recurrent state without entering any other state.

II.D. STEPPS system capabilities

The STEPPS system is designed for interactive use. It contains facilities to enter, manipulate, display, save and retrieve the description of a model. There are facilities to test the legality and consistency of a description. There is a facility for the automatic recognition of possible deadlocks. In addition, the STEPPS system contains a parameterizable model simulator and facilities to display or copy the data gathered during a simulation.

The notation defined earlier in this chapter is used both to enter a model description and to display the model. The displays available include processes and link connections, and transition values for a port and for a process. All possible paths between any two processes can be displayed, but this is a very expensive operation and not recommended because of large memory requirements.

Another feature of the STEPPS system is that it has been designed to facilitate application of analysis programs that might be defined externally to the STEPPS system.† Such analysis programs could be written in Sail [VanLehn 71], Bliss/10 [Wulf 71], or FORTRAN. These programs will be able to operate on process transition matrices, on process connection matrices, and on the graph connection matrix. The incorporation of externally defined functions necessitates a reconfiguration (a new LOAD) of the STEPPS system, but no program modifications are required.

---

†The details of doing this are presented in Appendix A.

Chapter III

The Use of the STEPPS Approach to Program Design

This chapter presents examples using the STEPPS model and the STEPPS system. These examples demonstrate that the model is rich enough to represent several standard program communication structures. One example demonstrates how the STEPPS system can be used in the initial design of a program and another example demonstrates how STEPPS can be used to analyze and help tune a multiprocessor program that is under construction and was designed without using the STEPPS system.

III.A. Use of the STEPPS model

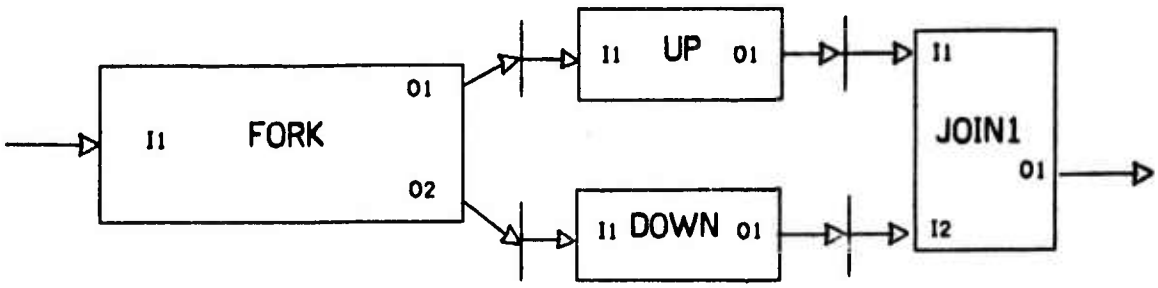
The STEPPS modeling schema can be used to represent a variety of program constructs. The program designer controls the amount of detail he wishes to include in a model. Since the STEPPS model has been shown to be able to represent the detail of both the UCLA and Petri net models, it can be used to represent programs at the same operation level as those models. However, STEPPS is intended to be used to depict a decomposition at a more modular level which more closely represents a functional system decomposition.

As a consequence of the STEPPS model being a communications structure model, programming details such as specific data dependent branching, indefinite (but finite) looping, case statements, and assignment statements are not intended to be modeled. Thus the following examples will demonstrate that the STEPPS model

abstraction, which is very much less expressive (or powerful) than a programming language and more expressive than the simpler Petri-net or UCLA model, has the expressive richness to model some real program structures.

III.A.1. Fork and join

Informally, the ability for the STEPPS model to represent multiple data paths has already been demonstrated. The situation is that one process can cause more than one other process to commence processing (Conway's "fork" [Conway 66]). After some concurrent processing, the data paths may unite and processing again occurs in only one processing unit (Conway's "join"). There are several ways to model fork and join. Figure III-1 shows one method. Process FORK sends a message to both process UP and process DOWN. In turn, they send messages to process JOIN1. JOIN1 must receive a message from UP before it requests a message from DOWN.




```
FORK.I1=    * O1:1.0,t          JOIN1.I1=    * I2:1.0,t
FORK.O1=      O2:1.0            JOIN1.I2=      O1:1.0,t
FORK.O2=      I1:1.0            JOIN1.O1=      I1:1.0

UP.I1=      * O1:1.0,t          DOWN.I1=     * O1:1.0,t
UP.O1=        I1:1.0            DOWN.O1=       I1:1.0
```

Figure III-1. Fork and join processes

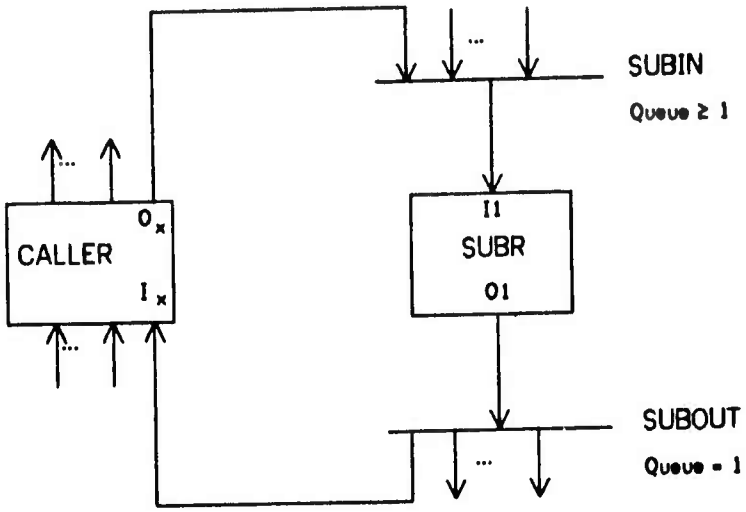
III.A.2. Subroutine processes

A subroutine process is a program that can be shared among several different processes. In terms of a STEPPS model a subroutine is a process that accepts messages from another process, performs some computation and then sends a message back to the calling process. Since messages do not contain any identification nor any other information, the subroutine can not direct a resulting message back to the caller process. Instead a technique is used whereby the caller waits for a response from the subroutine before it proceeds. Figure III-2 shows a graphical representation of the subroutine SUBR and the process, CALLER, that calls the subroutine. CALLER calls SUBR by sending a message to link SUBIN. As soon as the message is accepted CALLER waits for a reply from link SUBOUT. Within this model each process that wants to use the subroutine waits its turn to send a message to SUBR. Once the process sends its request to the subroutine it waits for a reply from the link SUBOUT. The timing parameters of the subroutine and the caller represent the action of a caller that does no processing concurrently with a subroutine process.†

A subroutine process can also be called while the caller process continues processing concurrently. This situation is modeled slightly differently than the one above. The difference is due to the requirement that the caller process receives the reply corresponding to its original request. Otherwise a second process could receive a reply before the subroutine computes long enough to request its processing, i.e. the second process receives the reply corresponding to the first process initialization. This problem exists because messages (as defined in the STEPPS model) do not contain

---

†An implication of this method is that there is no guarantee that the calling process receives the result of its call to the subroutine. However if, by convention, all calling processes take no time before requesting their respective results, no problem ensues because requests to and from the subroutine will occur in the same order.



CALLER.O_x = I_x:1.0,0.0 ! wait for response, no concurrent computation

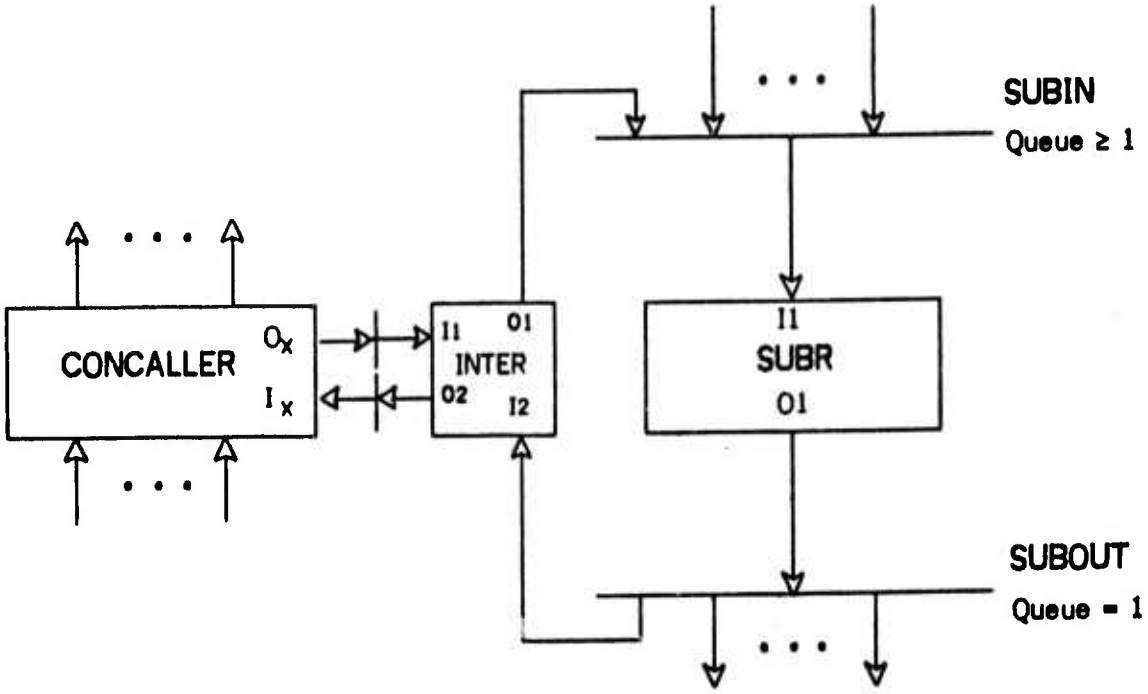
SUBR.I1 = O1:1.0,t ! t is subroutine compute time
SUBR.O1 = I1:1.0,0.0 ! Wait to be called again

Figure III-2. Subroutine process

information and processes do not direct messages to other processes, only to connected links. A solution is to introduce an intermediate process whose only function is to call the subroutine and wait for a response. This is shown in Figure III-3. CONCALLER continues to process before eventually requesting a reply from port I_x. It is necessary that O_x be *onto* I_x, i.e., once a message is sent from port O_x eventually a message will be requested at I_x. The process INTER will actually perform the subroutine call in the same manner as shown in Figure III-2.

III.A.3. Poisson processes and general service time processes

Typically, queueing theory models contain assumptions concerning the flow of messages within a system of message processors. These assumptions concern



CONCALLER.O_x = I_x:p,t ! p ≤ 1

INTER.I1 =* O1:1.0
INTER.O1 = I2:1.0
INTER.I2 = O2:1.0
INTER.O2 = I1:1.0

Figure III-3. Concurrent processing subroutine call

processing rates and take the form of assigning processing time as a random variable. The STEPPS system models a single processing time related to a given state, but it is possible to approximate a processing rate taken from some probability distribution.

The method used to approximate the production of messages with an interarrival rate taken from a known probability distribution is as follows:

1. Let f be the probability density function of the given distribution. Choose n to be the granularity of the approximation.
2. Divide the range of f into n distinct intervals $I_1, \ldots, I_n$.
3. $p_I = \int_{II} f(t)dt$ This is the probability of t being in the interval.
4. $t_I = (\int_{II} tf(t)dt)/(p_i)$

This is the expected value of t in the interval.

5. Let the process POISSON send messages to link LINK and form the connections LINK←POISSON.O1, ..., LINK←POISSON.O_n. See Figure (III-4).

6. The transition matrix for process A is defined by:

   POISSON.O_x = O_i: p_i, t_i for x, i= 1, ..., n.

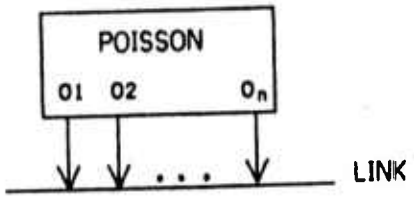


Figure III-4. Poisson arrival process

An example of this technique can be used to approximate a Poisson arrival rate in the following manner. The density function for an exponential rate between sending messages with mean λ is $(1/\lambda)e^{-t/\lambda}$

Choose a value, ε, for the probability of the start of the distribution. Thus

$$\varepsilon = \int_t^{\infty} (1/\lambda)e^{-t/\lambda}\, dt$$

which implies that the maximum value for t is $t_{max} = -\lambda \ln(\varepsilon)$. The interval $[t_{max},\infty)$ is one interval and that the remaining interval, $[0,t_{max})$, is divided into n-1 other intervals. For convenience, the division will be into uniform intervals of size $w = (t_{max})/(n-1)$.† Thus the values for the probabilities for intervals I_1 through I_{n-1} are:

$$p_i = \int_{i*w}^{(i-1)*w} (1/\lambda)e^{-t/\lambda}\, dt = e^{-(i-1)*w/\lambda} - e^{-i*w/\lambda}.$$

The values for the times for intervals I_1 through I_{n-1} are:

$$t_i = \int_{i*w}^{(i-1)*w} (t/\lambda)e^{-t/\lambda}\, dt = ((i-1)*w+\lambda)\, e^{-(i-1)*w/\lambda} - (i*w+\lambda)\, e^{-i*w/\lambda}.$$

When λ = 260, ε = 0.001, and n = 10, the values for p_i and t_i are:‡

†An example of a non-uniform interval will be shown in a later section of this chapter.

‡The STEPPS system has a feature to automatically determine the probability values and time given these parameters.

$A.O_x$ = O1: .535, 87.138; O2: .249, 286.696; O3: .115, 486.253; O4: .054, 685.81; O5: .025, 885.368

$A.O_x$ = O6: .016, 1084.925; O7: .005, 1284.482; O8: .002, 1484.041; O9: .001, 1683.604; O10: .001, 2056.011

In a similar manner, any arrival rate at a link (e.g. to link ALPHA above) can be approximated using the STEPPS model.

A general service time process can also be approximated using the STEPPS model (Figure III-5). The transition matrix for the general service time process, GENERAL, is defined by:

GENERAL.I1 = O_i: p_i, t_i and GENERAL.O_i = I1: 1.0, 0.0

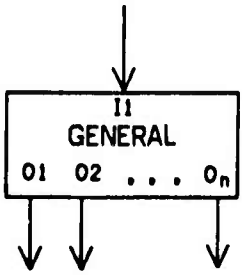


Figure III-5. General service time process

III.A.4. Pipeline of processes

One convenient structure for asynchronous multiprocessing is a pipeline consisting of a set of processes organized so that the results of one process form the data for the next process. Multiprocessing occurs when there are data in each of several processes in the pipeline. Figure III-6 shows the general structure of a pipeline of processes. At one end is a source of data units (process A) and at the other end is a sink for processed data units (process F). Connected in between the

two are processes each of which has input ports all attached to one link and output ports all attached to another link. Since the results of one process are the data for the next, each link between processes is connected to input ports of one process and output ports of a second process. Historically, structures similar to a pipeline have been successfully studied using queueing models [Kleinrock 75]. A STEPPS model obtains results pertaining to this structure by means of simulation.

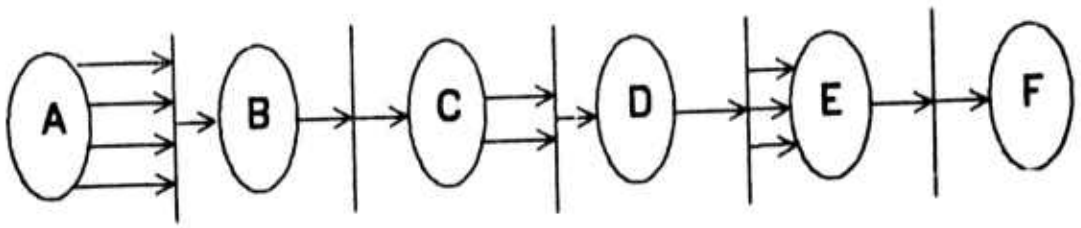


Figure III-6. Pipeline of processes

III.A.5. Synchronization

A multiprocessing program may contain process configurations that require synchronization. One of the better-known synchronization primitive sets is Dijkstra's P and V operations on a semaphore. It is possible to model this behavior with the STEPPS model. A process implements a P operation by sending a message to a LOCKSEM link and then waiting for a responding message before continuing (i.e. wait for a response from LOCKEDSEM link). Likewise a V operation corresponds to sending a message to an UNLOCK link. The STEPPS model would be:

```
LOCKSEM←PROCESS.0100          ! Attach to the lock semaphore
PROCESS.1100←LOCKEDSEM
PROCESS.0100= 1100:1.0        ! After performing a lock,
                              ! wait for a response before continuing.
PROCESS.1100= other ports
UNLOCKSEM←PROCESS.0101        ! Attach to unlock semaphore
```

The notational definition of the synchronization processes is as follows:

```
LOCKPROCESS.I1←LOCKSEM
LOCKPROCESS.I2←SEMAPHORE
LOCKEDSEM←LOCKPROCESS.O1
LOCKPROCESS.I1 = *I2:1.0          ! Obtain message from semaphore
LOCKPROCESS.I2 = O1:1.0           ! Let process performing lock continue
LOCKPROCESS.O1 = I1:1.0           ! Wait for next lock request
UNLOCKPROCESS.I1←UNLOCKSEM
SEMAPHORE←UNLOCKPROCESS.O1
UNLOCKPROCESS.I1 =* O1:1.0        ! Add one to semaphore
UNLOCKPROCESS.O1 = I1:1.0         ! Wait for more unlocks
Attributes SEMAPHORE Queue:n, Volume:1
                    ! n is maximum value for semaphore
                    ! Initial volume of 1 allows first lock to get through.
```

This technique is almost an exact analogy to Dijkstra's semaphores in that the number of messages residing in the SEMAPHORE link determines the number of LOCK operations that can be performed. The difference is that there is a limit, *n*, of possible locks. The use of UNLOCKPROCESS and UNLOCKSEM link is redundant. The process could be attached to SEMAPHORE instead of UNLOCKSEM:

```
SEMAPHORE←PROCESS.0101            ! Attach to unlock the semaphore.
```

The graphic structure of the lock/unlock processes is shown in Figure III-7.

A second example of a synchronization problem is the Reader/Writer problem. The problem is to allow multiple reader processes to be able to pass through a lock, but to exclude all writers so long as any reader is not complete. Once a writer process tries to perform a lock other readers and writers are not permitted until after the writer has performed an unlock. Naturally, the writer process does not proceed until all readers have completed their read unlocks. The solution to this problem requires three processes: READLOCK, WRITELOCK, and WRITEUNLOCK (Figure III-8). A reader process will send a message to the READLOCK process and wait for a reply from the READLOCKED link before continuing. Likewise, a writer process will send a message to the WRITELOCK process and wait for a reply from the WRITELOCKED link before continuing. The link RWLINK initially contains N messages. Each reader will

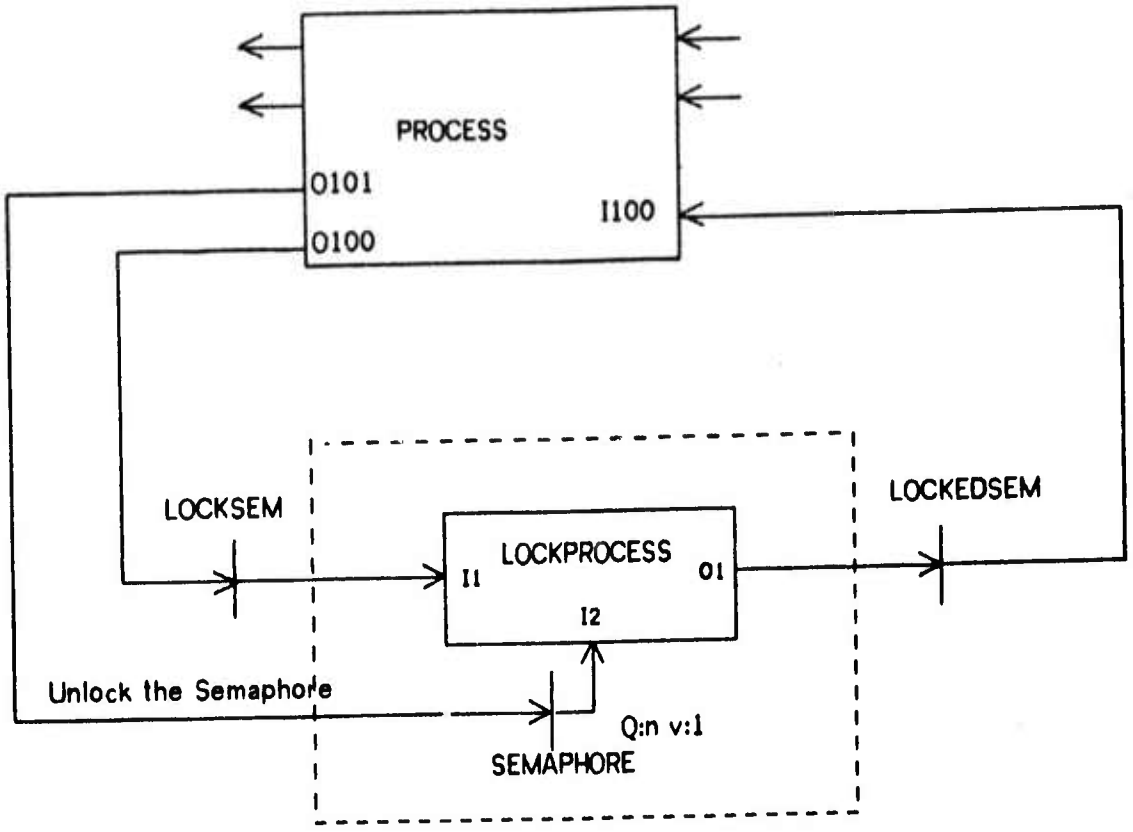


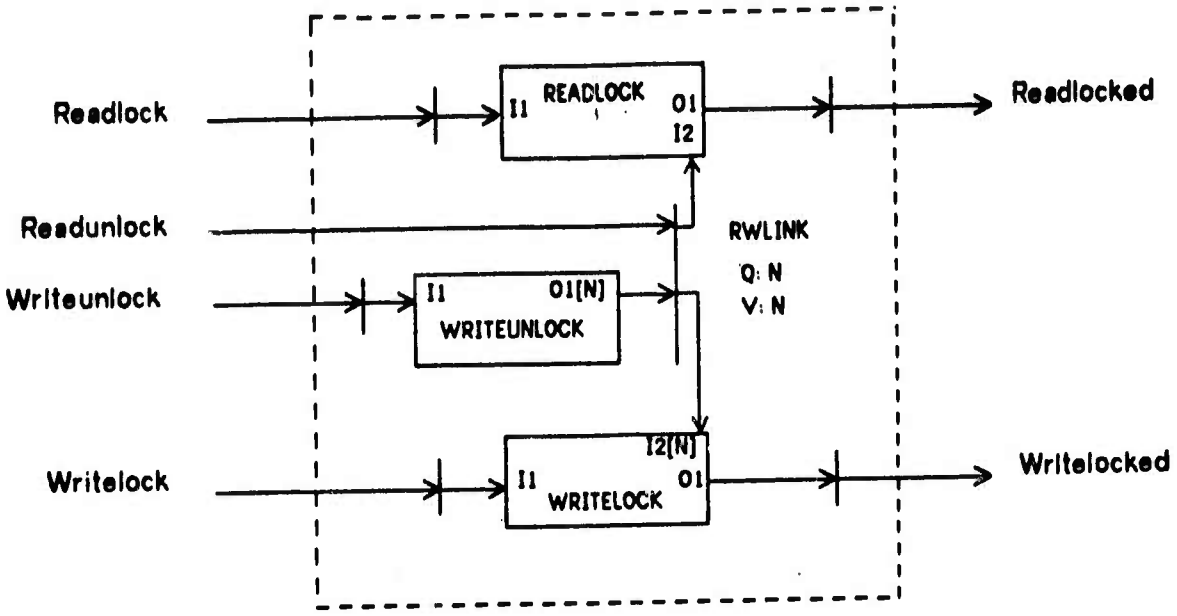
Figure III-7. Lock/Unlock synchronization

cause one message to be removed from RWLINK. Thus there can be a maximum of N simultaneous readers before any reader is blocked.† The WRITELOCK process requests all N messages from RWLINK before it allows a writer process to continue. When there are already reader processes that have requested messages from the RWLINK, WRITELOCK will wait until all current readers have performed a read unlock by sending a message to RWLINK (each such message will be requested by WRITELOCK). After WRITELOCK has all of the messages that were at RWLINK it allows a writer process to proceed. No readers can proceed since there will be no messages at RWLINK until a write unlock is performed by causing WRITEUNLOCK to send N messages to RWLINK

---

†As with the PV model, there is only a finite number of possible readers. This is not a problem because the STEPPS model does not include dynamic creation of processes.

and service to processes awaiting at a link is FIFO. Only one writer process can pass the lock since each would cause N messages to be requested from RWLINK and there can never be more than N messages there.




```
READLOCK.I2←RWLINK
WRITELOCK.I2←RWLINK←WRITEUNLOCK.O1

READLOCK.I1=        * I2:1.0      ! Request a message from RWLINK
READLOCK.I2=          O1:1.0      ! Allow a reader to proceed
READLOCK.O2=          I1:1.0      ! Wait for another read lock

WRITELOCK.I1=       * I2:1.0      ! Request message from RWLINK
WRITELOCK.I2[N]=      O1:1.0      ! After requesting N messages
                                  ! from RWLINK, allow a reader to proceed
WRITELOCK.O1=         I1:1.0      ! Wait for another write lock

WRITEUNLOCK.I1=     * O1:1.0      ! Send messages to RWLINK
WRITEUNLOCK.O1[N]=    I1:1.0      ! After sending N messages to
                                  ! RWLINK, wait for next writer unlock

Attribute RWLINK  Queue:N, Volume:N
```

Figure III-8. Reader/Writer Synchronization

III.B. Using STEPPS during system design: A Bliss/11 compiler

Bliss/11 [Wulf 72a] is a system implementation language designed for the DEC PDP-11 computer. Its only compiler is an optimizing cross compiler implemented on the DEC PDP-10. The language has been used as the implementation language for the Hydra operating system [Levin 75, Wulf 75b] for the C.mmp computer, as well as for other PDP-11 systems programs.

There are several reasons why a Bliss/11 compiler is an appropriate program to implement on a multiprocessor (C.mmp). First, since Bliss/11 is the system language for Hydra and C.mmp, it should be available on the Hydra system to make C.mmp self-sufficient. In addition, the mechanism for moving programs between the two computers is a time consuming and, presently, awkward arrangement. A second reason is that the Bliss/11 compiler is very large and slow. The compiler requires a large amount of PDP-10 memory to do even small compilations. A third reason is that the internal structure of the Bliss/11 compiler [Wulf 75a] consists of separate phases that could possibly be divided into separate processes†. Thus a Bliss/11 compiler is a program that can be considered for implementation on a multiprocessor.

The STEPPS system will be used to predict how a Bliss/11 implementation might perform as a multiprocess program. Possible structures for the compiler and structural refinements will be discussed.

III.B.1. An overview of the structure of Bliss/11

The Bliss/11 compiler is divided into seven relatively independent phases (Figure III-9). The L_i's in the figure refer to intermediate representations of data passed between the phases.

---

†This conjecture has been discussed with the authors of Bliss/11 [Wulf 75a].

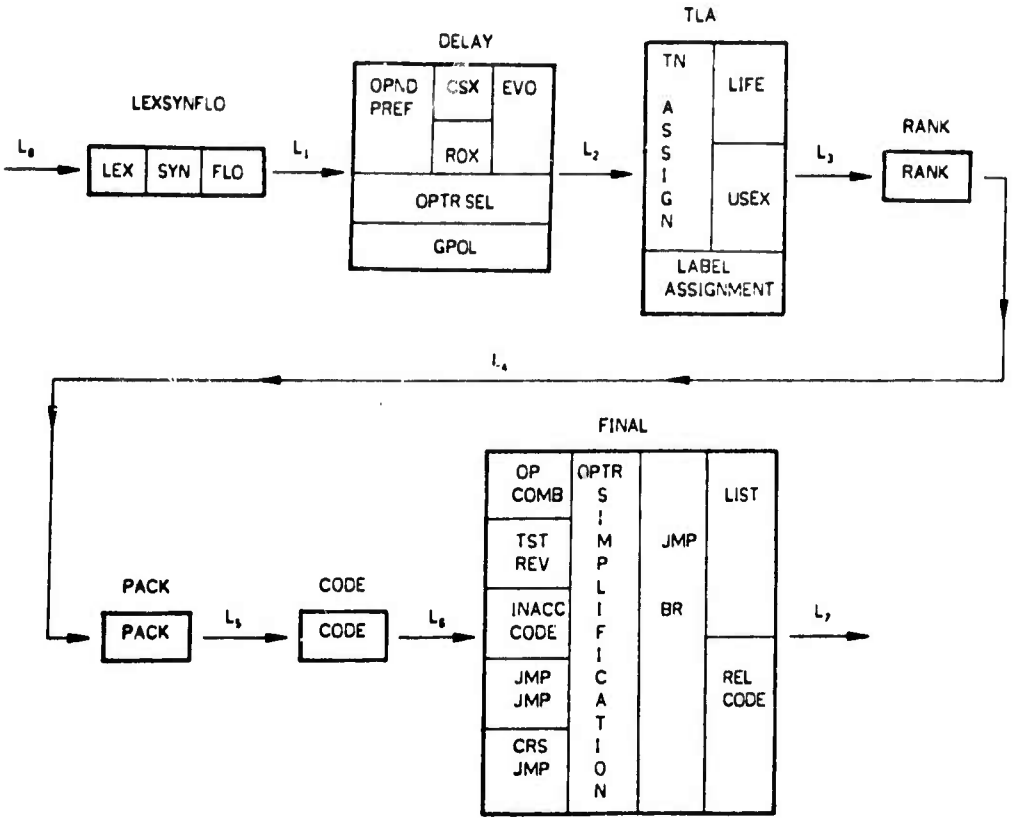


Figure III-9. Bliss/11 phase structure

The following is a description of the compiler [Wulf 75a][$]:

> . . . the subroutine is the program unit to which each physical phase is applied. Thus the source text for an entire subroutine is read and the phase LEXSYNFLO applied to it, producing intermediate form L_1. In turn DELAY, TLA, ..., and FINAL are applied to the intermediate representations L_1, L_2, ..., L_6 for the same subroutine, producing, respectively, L_2, L_3, ..., L_7. The next subroutine is processed only after all phases have been applied to its predecessor. A consequence of choosing the subroutine as the unit to which successive phases are applied is that optimizations are applied to this unit; i.e., no optimizations are applied which involve detailed structural knowledge of more than one subroutine simultaneously.
>
> The general attributes of the major phases are summarized below . . .
>
> LEXSYNFLO — This phase performs lexical analysis, declaration processing, syntax analysis, and flow analysis. The input is the source program unit in character string form. The output consists of: (1) a set of symbol table entries, (2) a tree representation of the parsed program unit, and (3) a set of lists (generally threads running through the tree) which define feasible global optimizations (constant expressions which may be moved out of loops and the like).
>
> DELAY — Delay has three primary functions: (1) to determine the "general shape" of the object code to be generated, (2) to estimate the "cost" of each (linear) program segment, and (3) to determine the evaluation order for expressions. By the "general shape" of the object code, we mean those properties of the operators (e.g., commutativity) or properties of the target machine (e.g., indexing) which may be used to simplify the computation of a value. Decisions are also made at this point as to whether any (or all) of the "feasible" global optimizations are, in fact, desirable. Actual machine code is not generated; rather various flags and fields are set to guide local code generation in a later phase. The cost metric is used to guide selection of evaluation order and in register allocation. The output of this phase is identical to that of LEXSYNFLO (i.e., symbol table, tree, etc.) except that certain information has

---

[$]Reproduced with permission.

been added to the tree to signal the subsequent phases of the compiler concerning the shape, cost, and execution order of the code to be generated.

TLA, RANK, PACK
: The function of these phases is what in other compilers is frequently called "register allocation"; the difference being that not only registers are allocated, but memory locations as well. The entities which are assigned to locations (registers or memory) include both compiler-generated temporaries and user-defined "local" variables. The output of this phase includes that of DELAY plus the bindings.

CODE
: The function of the CODE phase is to produce locally optimal code for each tree node; hence its output is a representation of the target machine language (the tree is discarded at this point). In some cases the locally optimal code is completely determined in DELAY; in these cases the action of CODE is trivial. In many cases, however, further analysis is required. For example, it is CODE's responsibility to determine the optimal sequence of shift and mask instructions to move an arbitrary subfield of one word into an arbitrary position of another.

FINAL
: FINAL has two responsibilities. The simpler of these is to prepare the final listing and object code files. The more interesting responsibility is a collection of relatively ad hoc "peephole" optimizations. These optimizations are performed by examining the actual code produced by CODE and eliminating inefficiencies which CODE was unable to detect. For example, FINAL will replace a jump instruction which transfers to another jump by one which transfers directly to the ultimate destination. It will also remove unreachable code, reverse the sense of certain tests, combine some instructions, etc.

As can be seen from the above, the phases operate independently of each other with respect to each subroutine. Thus while one phase is working with one subroutine another phase can be compiling a different subroutine. The compiler looks very much like a pipeline.

III.B.2. Application of the STEPPS system to Bliss/11

A multiprocess model of the Bliss/11 compiler was examined using the STEPPS system. A protocol of the use of the system for this application is presented in Appendix B. The issues that were explored concerning the multiprocess decomposition are:

1. How do specific alternate multiprocess decompositions of the compiler affect throughput? Throughput was measured in terms of the number of routines† processed per unit time.

2. Does the performance of the model suggest other decompositions?

3. When the number of processors is restricted, what are the effects of different scheduling algorithms?

4. What are the relationships among the number of processors available, the average number of active processes, and throughput?

The model of a multiprocess Bliss/11 compiler follows the same general pipeline structure as the phases of the original compiler [Wulf 75a]. Each phase is modeled as a server with an exponentially distributed processing rate.

Measurements of the operation of the real Bliss/11 compiler were taken; nine programs of differing complexity were compiled by an instrumented version of the actual compiler. The total time spent in each phase was determined and the corresponding percentage of total processing time was computed. These data are shown in Figure III-10. The phases are grouped slightly differently than those discussed earlier, due to actual Bliss/11 structural properties; LEX is separated from SYNFLO, and TNBIND combines TLA, RANK and PACK.

The processing rates of the STEPPS-modeled processes were chosen based on the percentage of total processing from the Bliss/11 measurements. For example, the processing rates for CODE and SYNFLO were chosen to be .084 units and .216 units

---

†The unit of compilation in the Bliss/11 compiler.

Phase	Time (seconds)	Percent of Total
LEX	67.92943	26.0 %
SYNFLO	56.38539	21.6 %
DELAY	9.64012	3.7 %
TNBIND	2.78647	10.7 %
CODE	22.08524	8.4 %
FINAL	77.17126	29.6 %
Total	261.07621	100.0 %

Figure III-10. Bliss/11 measured data

respectively. The LEX process was considered to be the generating process which provided elements to be processed at an exponential rate with mean .260 units. Figure III-11 shows the set of commands to the STEPPS system used to create the model (Appendix A contains a complete description of the STEPPS commands).

```
Model BL11
Density expon port lex.o0 link ls mean .26
Density expon port synflo.o0 link sd mean .216
Density expon port delay.o0 link dt mean .037
Density expon port tnbind.o0 link te mean .122
Density expon port code.o0 link cf mean .084
Density expon port final.o0 link fr mean .296
synflo.I20←ls
delay.I20←sd
tnbind.I20←dt
code.I20←tc
final.I20←cf
synflo.o0 = I0:0; I20:1/I0:1
copy delay.I20, tnbind.I20, code.I20, final.I20.synflo.I20
copy delay.o0, tnbind.o0, code.o0, final.o0:synflo.o0
result.i0←fr
schedule noncompete result
attribute tc,cf,dt,fr,ls,sd      Queue:100
```

Figure III-11. STEPPS Bliss/11 model commands

A graph representation of this model is shown in Figure III-12.

The first set of experiments consisted of simulating the model with one to six

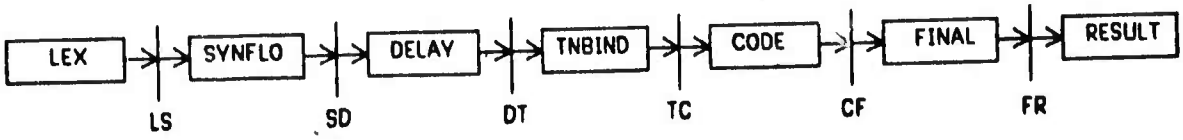


Figure III-12. Bliss/11 graph model

processors using one process per phase. For each number of processors, the effects of three scheduling algorithms were also measured. These algorithms were: First-In-First-Out (FIFO), Random, and Link (select the process with the largest number of waiting messages). These algorithms are discussed in Chapter V.

The results of these experiments are shown in Figures III-13, III-14, III-15, III-16, and III-17. The measurements were performed on 700-900 messages (representing routines) passing from the LEX process through the FINAL process. The maximum possible throughput rate per experiment (i.e., simulation execution) is the rate at which routines are produced by the LEX process. Thus the maximum expected throughput rate is the reciprocal of the processing rate of LEX for each simulation, 4.00 routines per unit time when the expected time between routines is .250 (1 processor, FIFO). The observed throughput rate was found by dividing the number of routines entering RESULT by the total processing time.

Prcrs.	LEX Rate	Thru Rate	% Thru Rate	Avg. Active	Avg. Waiting
1	.254	0.96	24.0	1.00	4.98
2	.272	1.78	48.4	2.00	3.28
3	.279	2.74	76.4	2.96	1.61
4	.252	3.40	85.6	3.33	0.44
5	.272	3.37	91.7	3.66	0.04
6	.259	3.28	88.3	3.57	0.00

Figure III-13. Bliss/11 Simulation FIFO Table

The measure that was used as the basis for comparing performance was the

Prcrs.	LEX Rate	Thru Rate	% Thru Rate	Avg. Active	Avg.Waiting
1	.248	0.99	24.6	1.00	4.98
2	.241	1.84	44.3	2.00	3.39
3	.271	2.65	71.8	2.65	1.83
4	.259	3.45	89.4	3.54	0.49
5	.255	3.58	91.3	3.57	0.00
6	.270	3.26	89.7	3.47	0.00

Figure III-14. Bliss/11 Simulation LINK Table

Prcrs.	LEX Rate	Thru Rate	% Thru Rate	Avg. Active	Avg.Waiting
1	.242	0.91	22.0	1.00	4.64
2	.241	1.96	47.2	2.00	3.37
3	.273	2.78	75.9	2.94	1.53
4	.263	3.36	88.4	3.50	0.48
5	.257	3.56	91.5	3.62	0.06
6	.259	3.50	90.6	3.64	0.00

Figure III-15. Bliss/11 Simulation RANDOM Table

percent of maximum throughput rate. This measure was chosen because the measured throughput rates varied due to the approximation to exponential processing rates. For example, four processors using FIFO scheduling showed a throughput rate of 3.40 out of max rate of 1/.252 = 3.97 for 85.6 percent.

Several implications concerning this multiprocess model were apparent from these results. First, the addition of more processors has a major, approximately linear, effect on throughput until four processors are used. Addition of a fifth processor does not cause a very large improvement (about 86% to 91%). Adding a sixth processor does not indicate any significant difference. Another factor is that the different scheduling algorithms do not seem to significantly affect the model's performance. The ***average number of active processes*** (and processors) and ***average number of ready processes*** measures also indicate that after four processors are available most of the

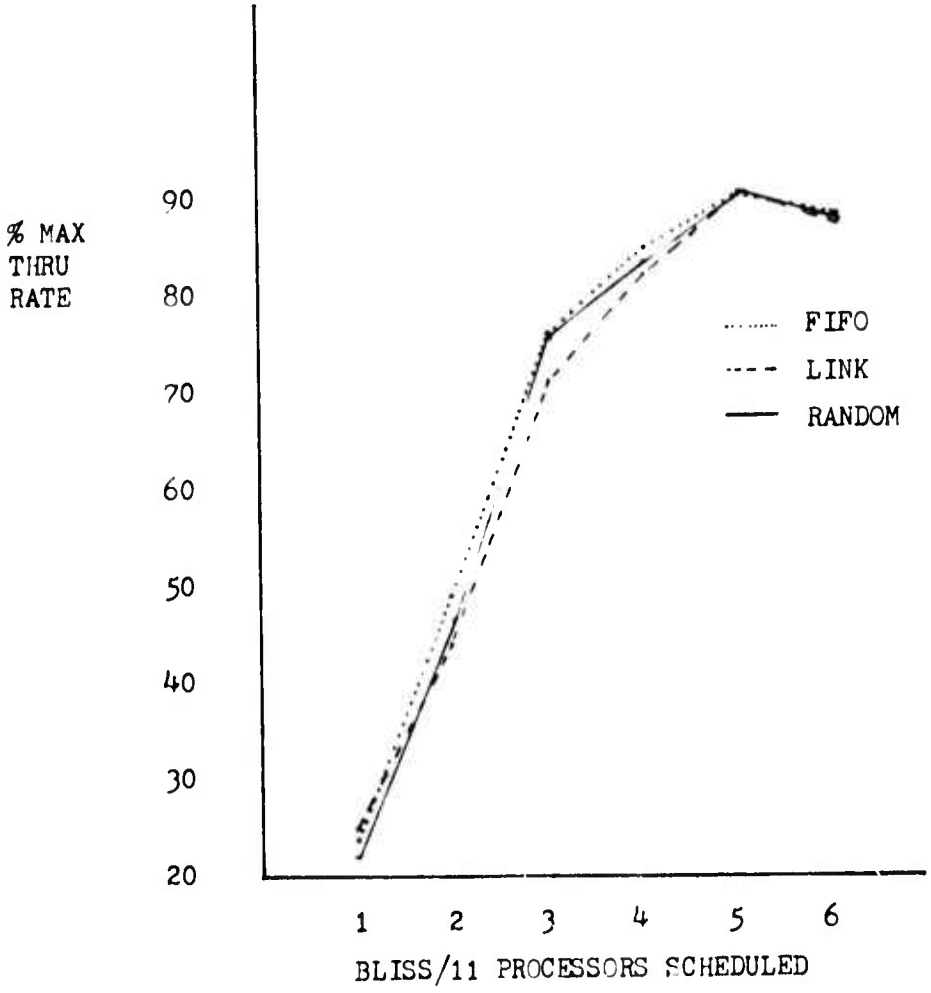


Figure III-16. Bliss/11 Percentage Maximum Throughput

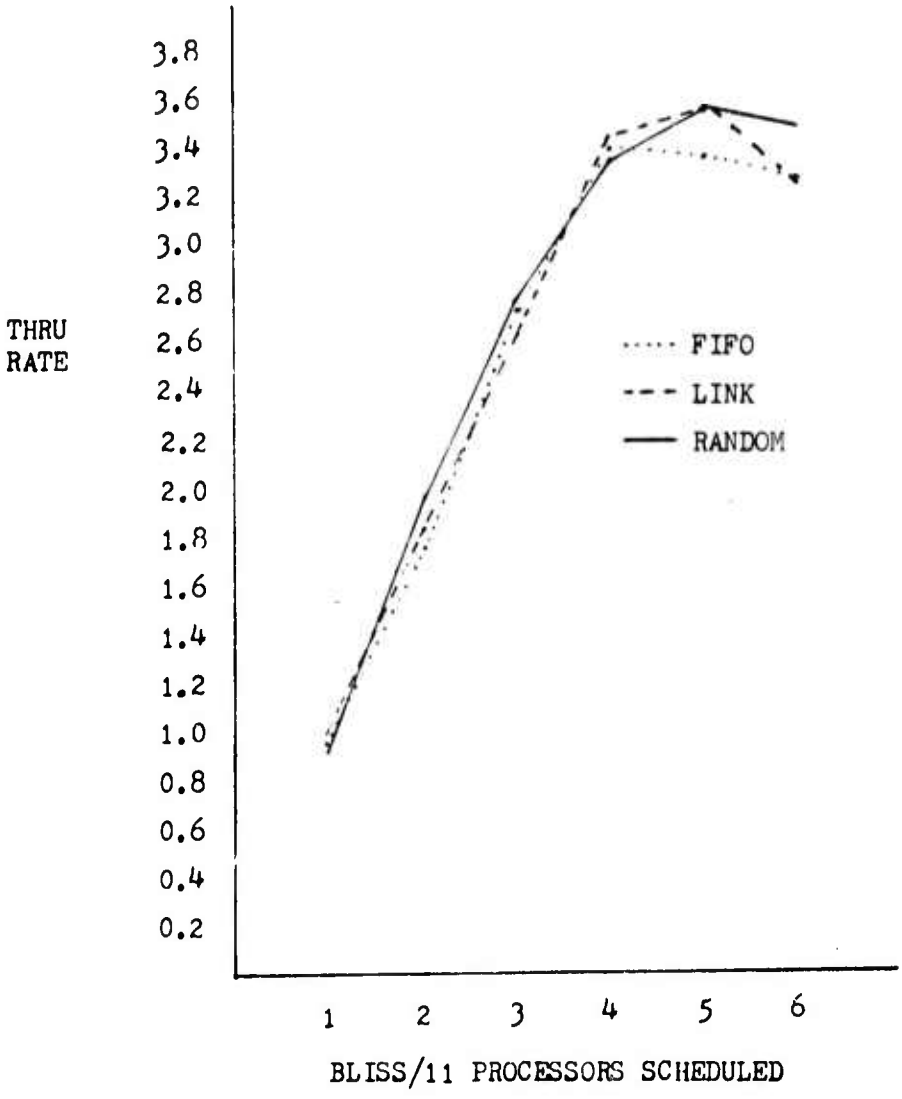


Figure III-17. Graph of Measured Throughput

required processing power is available. This helps confirm the observation that the addition of more processors beyond four does not lead to as major a performance improvement as adding one processor to fewer than four processors.

The next set of experiments included multiple copies of some of the slower processes as part of the model as an alternative to the simple pipeline structure. The only process that could not be duplicated was the LEX process since part of its function is recognizing the sequences of characters as delimiting a routine definition.† An examination of the data indicated that there were three major bottlenecks. The bottlenecks were identified by locating links between processes where the expected queue length was large. Figures III-18, III-19, and III-20 show the expected queue lengths at the links between the processes. Naturally, these were the same processes that had relatively slow processing rates‡. Three alternate structures were examined:

A. 2 FINALs and 2 SYNFLOs;

B. 3 FINALs and 2 SYNFLOs;

C. 3 FINALs, 2 SYNFLOs and 2 TNBINDs.

Prcrs.	LS	SD	DT	TC	CF
1	9.194	0.000	0.000	0.000	0.000
2	8.403	5.148	7.172	8.107	9.354
3	2.108	0.010	0.263	1.505	6.401
4	5.682	6.648	8.817	9.048	9.643
5	2.765	1.770	2.556	3.309	6.674
6	3.595	2.100	3.432	5.350	8.732

Figure III-18. Bliss/11 Simulation FIFO Queue Lengths

These structures were run with 1, 5, 8 and all possible processors. Although

---

†This can be done by Begin-End counts.

‡The large queue that formed before the Code phase was due to Code being unable to send results to Final and thus had to wait before processing new routines.

Prcrs.	LS	SD	DT	TC	CF
1	0.410	0.616	0.000	0.219	0.001
2	7.923	0.087	0.987	5.364	9.111
3	4.139	0.078	0.713	4.533	8.892
4	3.455	1.136	4.623	6.972	9.248
5	4.266	3.268	5.757	7.018	8.945
6	5.903	6.204	8.404	8.999	9.836

Figure III-19. Bliss/11 Simulation LINK Queue Lengths

Prcrs.	LS	SD	DT	TC	CF
1	8.461	4.329	1.796	0.623	0.717
2	7.928	0.404	0.864	1.362	7.925
3	4.561	0.047	0.975	5.494	8.[illegible]3
4	3.355	2.727	5.326	7.749	9.353
5	3.419	1.488	3.306	5.738	9.176
6	2.976	3.762	7.766	8.661	9.472

Figure III-20. Bliss/11 Simulation RANDOM Queue Lengths

the simulations were run using all three scheduling algorithms, there was not much difference in performance due to the scheduling algorithm (less than one per cent). Thus Figures III-21, III-22 and III-23 show the results using either FIFO or LINK scheduling. Figures III-22 and III-23 also show graphs of the FIFO results without using multiple copies of phases. As the graphs show, each of the multiple process per phase models performs better than the single process per phase model, given enough processors. Structure C, above, performed the best among them.

The difference among the structures was not very large, viz. about 5% of the maximum rate. Although there is improvement using the multi-copy structures, the improvement over the single process per phase does not appear to warrant such structure. Instead, the bottleneck appears to be the LEX process which is inherently sequential. This observation suggested another experiment to determine the effects of

Decom-position	Prcrs.	LEX Rate	Avg. Active	Avg. Ready	Thru Rate	% Max Thru Rate
B11 (A)	1	.253	1.00	4.78	0.98	24.8
	3	.251	2.88	2.23	2.87	72.0
	5	.272	3.68	2.28	3.59	96.8
	8	.267	3.79	0.00	3.70	98.8
B11 (B)	1	.276	1.00	4.79	0.98	27.0
	3	.276	2.82	2.04	2.77	76.4
	5	.261	3.71	0.38	3.65	95.3
	6	.255	3.99	0.00	3.89	99.2
	8	.253	3.94	0.00	3.93	99.4
	9	.278	3.64	0.00	3.58	99.5
B11 (C)	1	.288	1.00	5.43	0.92	26.5
	3	.263	2.81	2.25	2.76	72.6
	5	.274	3.55	0.41	3.46	94.8
	8	.259	3.86	0.01	3.85	99.7
	11	.244	4.03	0.00	4.07	98.3

Figure III-21. Table of Results of Multi-copy Bliss/11 Phase Models

further decomposing LEX into a pipeline of phases: FILE, ATOM, and NT|SEARCH. The goal was to increase the rate at which messages reached the SYNFLO phase. The results of this set of experiments are shown in the table of Figure III-24. It can be seen that the rate at which messages queued up to the SYNFLO phase decreased from .26 to .18 for an increase of 44% due to the further decomposition of the LEX phase.

Other process structures may also be studied using the STEPPS system. Current research into the phases of Bliss/11 indicates that two of the phases could be restructured. The DELAY phase [Johnsson 76] could perform more complex operations (and would be slower). The FINAL phase† could also be altered or decomposed even further into smaller independent processes.

Since the data presented represent about fifty separate model simulations, the Bliss/11 experiments were executed over several weeks. Each simulation required

---

†S. Hobbs, current research.

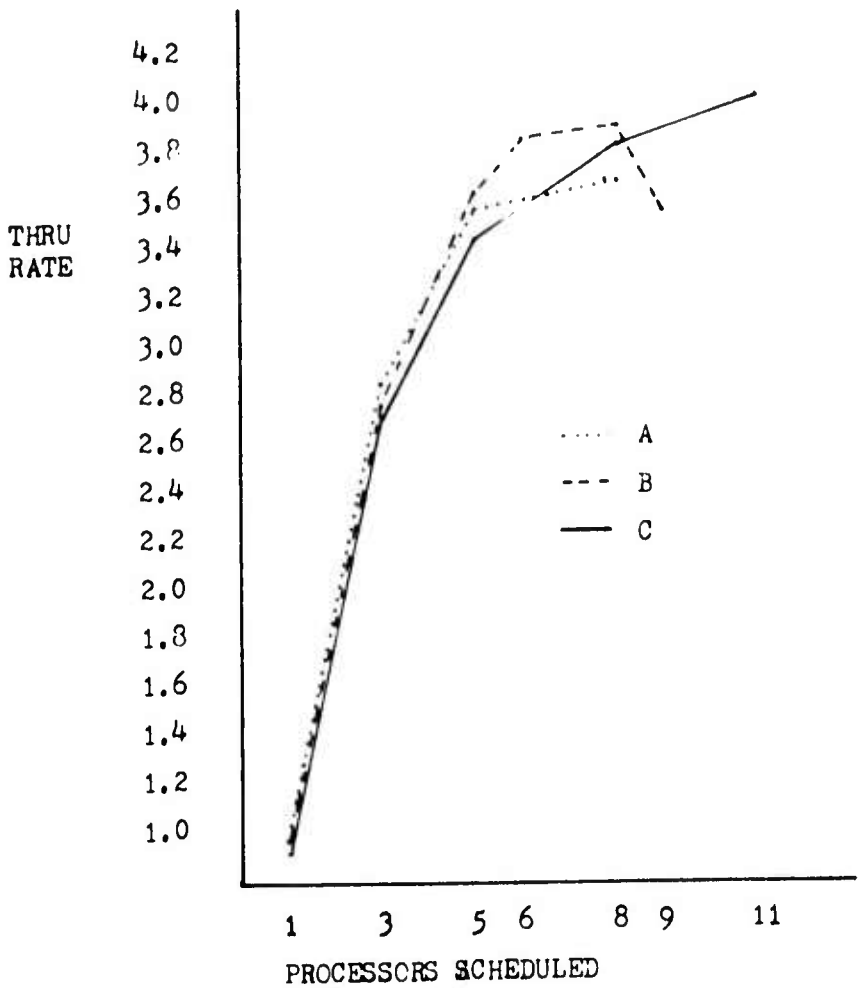


Figure III-22. Multi-copy Bliss/11 Phase Model Thru Rate Graph

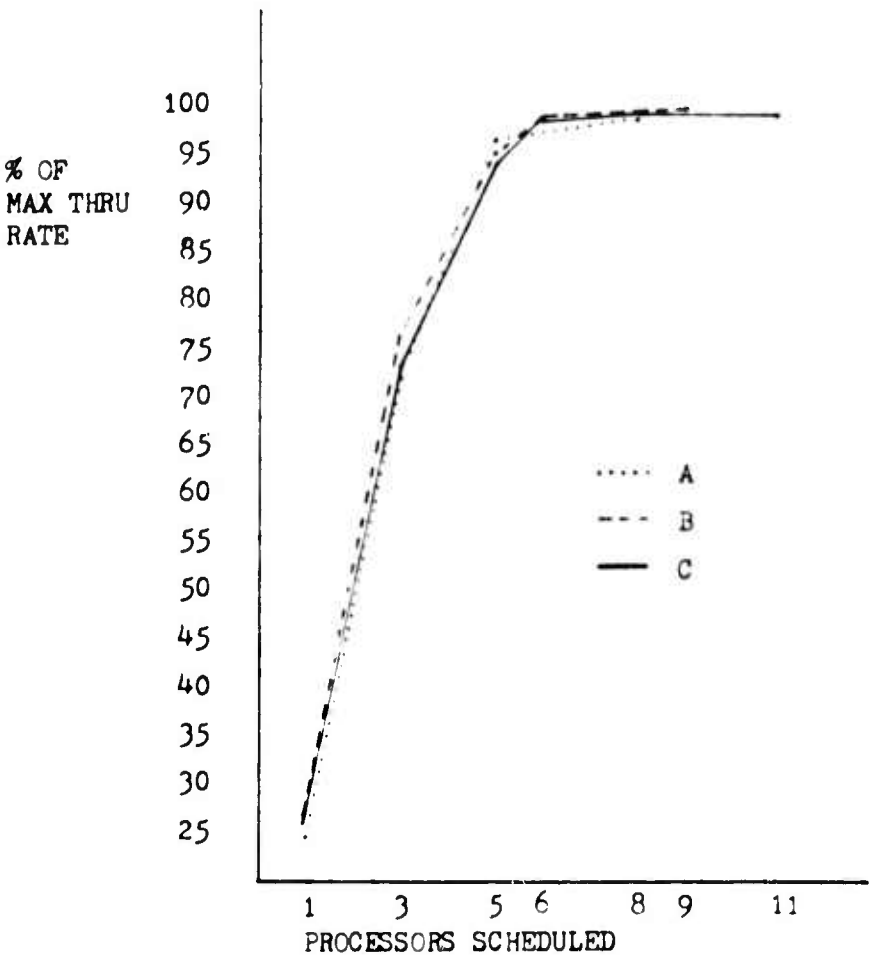


Figure III-23. Multi-copy Bliss/11 Phase Model Percentage Max Thru Rate Graph

Prcrs	FILE Rate	NS Send Rate	Avg. Active	Avg. Ready	Thru Rate	% Max of FILE	% Max "LEX"
3	.119	.29	2.96	2.97	2.38	27.13	69.02
5	.120	.20	4.41	0.79	3.20	38.40	64.00
7	.126	.18	4.92	0.02	3.52	44.35	63.36
8	.121	.18	4.85	0.00	3.29	39.81	59.22

Figure III-24. LEX Decomposition Results

from four to thirty minutes of execution time for a total of about six hours of execution time. This amount of time was not particularly large since it is about the same amount of time that was required to obtain the Bliss/11 data originally.

As detailed in Appendix C, these simulation experiments were statistically validated. Based on trial runs, message traffic flows and simulation run times were determined for eliminating initial condition bias in the subsequent experiments. Since there were many different simulation experiments, one was chosen for developing statistical confidence intervals. Thus, for the experiment using six processors and FIFO scheduling, the 90% confidence intervals computed were: LEX Computing Time, [.245,.264]; Percent Thru, [84.6,88.5]; and Thru Rate, [3.28,3.57]. Comparing these intervals with the results shown in Figure III-13, it can be seen that each of the values falls within these respective confidence intervals (i.e, 2.59, 88.3, 3.28).

These Bliss/11 experimental results should have several implications to system designers of a multiprocess Bliss/11 compiler. Foremost is the conclusion that there should be an increase in processing throughput of about four times over a sequential compiler. This estimated increase is significant in that it demonstrates both potential benefits and potential limitations in developing a (possibly) complex multiprocess Bliss/11 compiler. Given that the designer chooses to develop the multiprocess compiler, it can be observed that the compiler should not necessarily be designed to

dedicate a processor to each process. The simulated result shows that there is an approximate linear increase in throughput when using a small number of processors, but after about two thirds of the number of potential processors are used the maximum throughput rate is almost achieved. The bottleneck was shown to be the lexical analysis phase of the compilation process. Finally, it was shown that simple scheduling disciplines (FIFO and most messages waiting) did not affect potential throughput rate more than a random process scheduling technique. Thus these simple experiments using the STEPPS model and STEPPS system should provide information that would affect the design of a multiprocess Bliss/11 compiler.

III.C. Using STEPPS during system construction and tuning: Hearsay II

The Hearsay II speech understanding system (HSII) [Fennell 75a, 75b, Lesser 74] has been designed to utilize a variety of analysis sources to solve the problem of understanding human speech for performance of a task [Newell 71]. The problem has been functionally decomposed so that individual subparts of the problem solution can be performed concurrently, with each contributing to the speech understanding task.

The Hearsay II system is being implemented on both a uniprocessor, a DEC PDP-10, and in a similar form on a multiprocessor, the CMU C.mmp. The uniprocessor implementation is structured as if it were being implemented on a multiprocessor, with a scheduler deciding on the actual order of processing. The C.mmp implementation contains some design alternatives chosen to reflect restrictions due to the Hydra operating system [Levin 75, Wulf 74]. Some implementation issues are common for both machines since the systems are based on the same design.

III.C.1. Overview of Hearsay II system organization

The following is a description of the organization of the HSII system [Fennell and Lesser 75][†]

> . . . The Hearsay II speech-understanding system (HSII) (Lesser, *et al.* 1974; Fennell, 1975; and Erman and Lesser, 1975) currently under development at Carnegie-Mellon University represents a problem-solving organization that can effectively exploit a multiprocessor system. HSII has been designed as an AI system organization suitable for expressing ***knowledge-based problem-solving strategies*** in which appropriately organized subject-matter knowledge may be represented as ***knowledge sources*** capable of contributing their knowledge in a parallel data-directed fashion. A ***knowledge source*** may be described as an agent that embodies the knowledge of a particular aspect of a problem domain and is useful in solving a problem from that domain by performing actions based upon its knowledge so as to further the progress of the overall solution. It is felt that the knowledge source is an appropriate unit for use in the decomposition of a knowledge-intensive task domain. Knowledge sources, being suitably organized capsules of subject-matter knowledge, may be independently formulated as various pieces of the knowledge relevant to a task domain become crystallized. The HSII system organization allows these various independent and diverse sources of knowledge to be specified and their interactions coordinated so they might cooperate with one another (perhaps asynchronously and in parallel) to effect a problem solution. As an example of the decomposition of a task domain there might be distinct knowledge sources to deal with acoustic, phonetic, lexical, syntactic, and semantic information.
>
> * * *
>
> . . . A ***production system*** is a scheme for specifying an information processing system in which the control structure of the system is defined by operations on a set of ***productions*** of the form 'P' → A', which operate from and on a collection of data structures. 'P' represents a logical antecedent, called a ***precondition***, which may or may not be satisfied by the information encoded within the dynamically current set of data structures. If 'P' is found to be satisfied by some data structure, then the associated ***action*** 'A' may be executed, which presumably will have some altering effect upon the data base such that some other (or the same)

[†]Used with permission.

precondition becomes satisfied. This paradigm for sequencing of the actions can be thought of as a data-directed control structure, since the satisfaction of the precondition is dependent upon the dynamic state of the data structure. Productions are executed as long as their antecedent preconditions are satisfied, and the process halts either when no precondition is found to be satisfied or when an action executes a stop operation (thereby signalling problem solution or failure, in the case of problem-solving systems).

* * *

. . . The HSII system organization, which can be characterized as a "parallel" production system, has a centralized data base which represents the dynamic problem solution state. This data base, which is known as the ***blackboard***, is a multidimensional data structure which is readable and writable by any precondition or knowledge-source process (where a knowledge-source process is the embodiment of a production action). Preconditions are procedurally oriented and may specify arbitrarily complex tests to be performed on the data structure in order to decide precondition satisfaction. Preconditions are themselves data-directed in that they are tested for satisfaction whenever relevant changes occur in the data base, and simultaneous precondition satisfaction is permitted. Testing for precondition satisfaction is not presumed to be an instantaneous or even an indivisible operation, and several such precondition tests may proceed concurrently.

* * *

. . . The basic structure and components of the HSII organization may be depicted as shown in the message transaction diagram of Figure III-25. The diagram indicates the paths of active information flow between the various components of the problem-solving system as solid arrows; paths indicating control activity are shown as broken arrows. The major components of the diagram include a passive global data structure (the ***blackboard***) which contains the current state of the problem solution. Access to the blackboard is conceptually centralized in the ***blackboard handler*** module,† whose primary function is to accept and honor requests from

---

†The blackboard handler module could be implemented either as a procedure which is called as a subroutine from precondition and knowledge source processes, or as a process which contains a queue of requests for blackboard access and modification sent by precondition and knowledge source processes. In the implementation discussed in the paper (i.e., Fennell and Lesser 75), the blackboard handler module is implemented as a subroutine.

> the active processing elements to read and write parts of the blackboard. The active processing elements which pose these data access requests consist of ***knowledge-source processes*** and their associated ***preconditions.*** Preconditions are activated by a ***blackboard monitoring mechanism*** which monitors the various write-actions of the blackboard handler; whenever an event occurs which is of interest to a particular precondition process, that precondition is activated. If upon further examination of the blackboard, the precondition finds itself "satisfied," the precondition may then request a process instantiation of its associated knowledge source to be established, passing the details of how the precondition was satisfied as parameters to this instantiation of the knowledge source. Once instantiated, the knowledge-source process can respond to the blackboard data condition which was detected by its precondition, possibly requesting further modifications to be made to the blackboard, perhaps thereby triggering further preconditions to respond to the latest modifications. This particular characterization of the HSII organization, while certainly overly simplified, shows the data-driven nature of the knowledge source activations and interactions.

III.C.2. STEPPS model of Hearsay II organization

The STEPPS model was used to represent the operation of the individual processing components of the HSII system, the precondition (PC) processes and the knowledge source (KS) processes. In addition the data base (DB) blackboard was modeled as a set of synchronization locks similar to those presented in Section III.A.5. In some cases locks cascaded, i.e. a lock operation caused performance of two or more other locks. The details of the STEPPS HSII models are shown in Appendix B.

Figure III-26 shows a detailed description of the PC process actions and Figure III-27 shows the corresponding STEPPS graphic and system transition matrix notations. The essential common actions of a PC are modeled: wait for condition, examine DB, compute, possibly initiate a KS, and repeat.

Similarly, Figure III-28 shows a detailed description of the KS process actions and Figure III-29 shows the corresponding STEPPS graphic and transition matrix

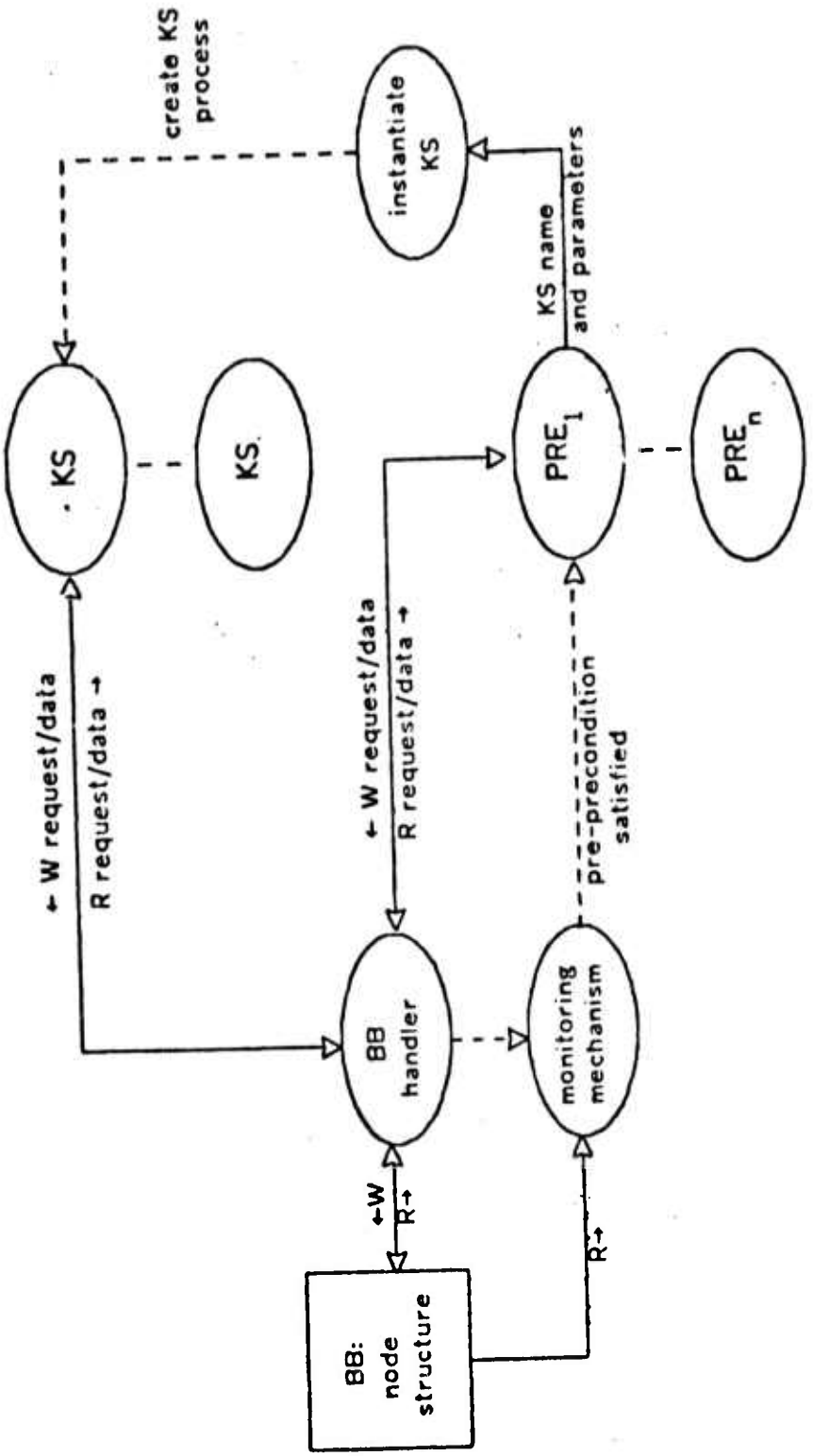
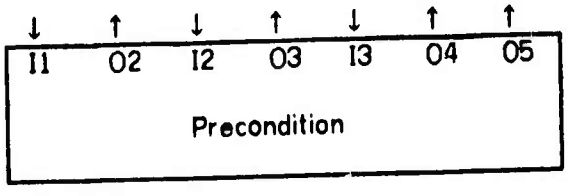


Figure III-25. Simplified HSII System Organization

Precondition

I1: Wait for condition occurrence
With probability p_c wait for more condition occurrences (go to I1)
O2: Perform DB read lock
I2: Wait for lock completion
O3: Perform read
I3: Wait for read completion
Compute
With probability p_r perform more reads (go to O3)
O4: Perform DB unlock(s)
O5: Start up a KS (or set of KS's or no KS's)
terminate processing (go to I1)

Figure III-26. Description of Precondition Process




```
PC.I1=*  I1:pc,tpc;O2:1-pc,t1-pc   ! Either wait for more messages
                                   ! or DB read lock
PC.O2=   I2:1.0                    ! Wait for lock completo
PC.I2=   O3:1.0                    ! Perform read
PC.O3=   I3:1.0                    ! Wait for read complete
PC.I3=   O3:pr,tpr; O4:1-pr,t1-pr  ! Either read more or unlock
PC.O4=   O5:1.0,t                  ! Start up KS(s); the time is processing time
                                   ! before restart
PC.O5=   I1:1.0                    ! Wait for restart
```

Figure III-27. STEPPS Precondition Model

notations. The essential common KS process actions are modeled: wait to start, examine data base, process, and possibly alter the data base.

It can be seen from these descriptions that there are relationships between the Precondition processes and the Knowledge Source processes. These are relationships whereby PC's send messages to KS's. In STEPPS, this is represented by:

```
KS.I1←KSLINK←PC.O5          ! Connect PC to KS through KSLINK
```

Knowledge Source

I1: Wait for wake up
O2: Perform DB read lock(s)
I2: Wait for lock completion(s)
O3: Perform read
I3: Wait for read completion
Compute
With probability p_r perform more reads (go to O3)
O4: Perform DB read unlock(s)
Compute
With probability p_t terminate processing (go to I1)
O5: Perform DB write lock(s)
I5: Wait for lock completion
O6: Perform write
I6: Wait for write completion
Compute
With probability p_w perform more writes (go to O6)
O7: Perform DB write unlock(s)
Terminate processing (go to I1)

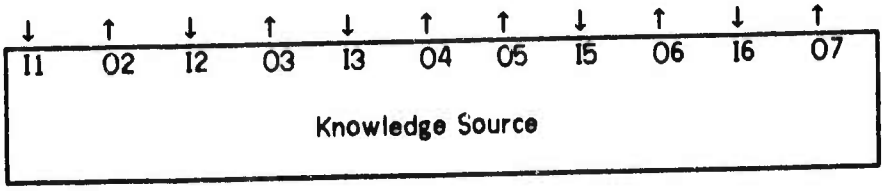
Figure III-28. Knowledge Source Process Description

The model has been designed so that there is some decision process which chooses which PC will next receive notice to start processing. This decision process, called PCSELECTOR, is attached to the port I1 of each precondition. Figure III-30 shows the graphical relation between PCSELECTOR and the set of preconditions. A possible transition matrix for PCSELECTOR when there are n preconditions is:

$$\text{PCSELECTOR}.O_x = O1{:}p_1;\ O2{:}p_2;\ \ldots\ O_n{:}p_n \quad \text{for } x=1,\ldots,n$$

The PC processes and KS processes interact with each other by reading and writing the data base. The data base accessing is an example of the Reader/Writer problem that was discussed in an earlier section.

The Hearsay II system has been designed to allow the dynamic creation of KS processes. These processes perform their respective operations and then disappear. Since the STEPPS model was not designed to allow for this facility, it must be




```
! Transition matrix                                   Next step
KS.I1=*   02:1,t1                                    ! t1 represents computation time before
                                                     ! doing read lock
KS.02=    I2:1.0                                     ! Wait for read lock completion
KS.I2=    03:1.0                                     ! Perform read
KS.03=    I3:1.0                                     ! Wait for read lock
KS.I3=    03:pr, tpr; 04:1-pr,t1-pr                  ! Either do more reads or perform unlock
                                                     ! The times can be different
KS.04=    I1:pt,tpt; 05:1-pt,t1-pt                   ! Either terminate or perform write unlock
KS.05=    I5:1.0                                     ! Wait for write lock
KS.I5=    06:1.0                                     ! Perform write
KS.06=    I6:1.0                                     ! Wait for write completion
KS.I6=    06:pw,tpw; 07:1-pw,t1-pw                   ! Either write more or unlock

KS.07=    I1:1                                       ! Wait to restart
```

Figure III-29. STEPPS Knowledge Source Model

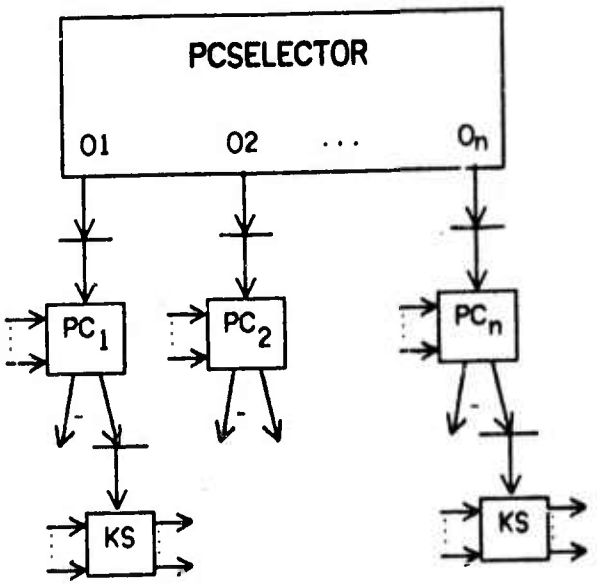


Figure III-30. PCSELECTOR process

approximated. The method is to allow a fixed number of instantiations of a single KS to act as a pool of KS's. These KS's compute in-parallel since different copies of the KS can accept messages from their entry link (Figure III-31). The model performs as if there were some maximum number of KS's of each type allowed. When a suitable number of copies of a KS are available the limit will not affect performance.

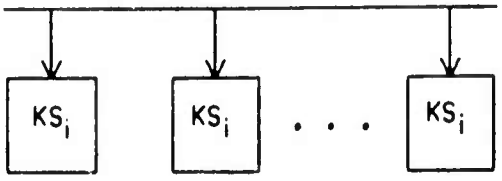


Figure III-31. Set of identical Knowledge Sources

III.C.3. Performance questions pertaining to the HSII model

The model of HSII emphasizes implicit interprocess communication via data directed processing. This communication is the basis for interprocess interference which occurs either when processes are blocked when attempting to perform a data lock or when a process waits for the occurrences of actions of another process (modeled as waiting for a message).

The following are pertinent questions for structuring of the Hearsay II system:

1. How much of the data base is locked and when?
2. What is the expected interference due to the locking?
3. How do various locking strategies compare?
4. Should a PC start up a set of KS instantiations sequentially, in parallel or in groups?
5. How many processors are needed?
6. What are the effects of alternate scheduling algorithms?

7. How can the processing load be balanced among available processors and with respect to the data base?

8. Is there a particular number of processes that should be dedicated to KS's and another number that should be dedicated to PC's?

The ultimate goal is to be able to solve the speech problem in the least amount of real time. The questions relate to the goal in that they provide an understanding of those places where Hearsay II is performing well and poorly with respect to interprocess activity.

III.C.4. Application of the STEPPS system to Hearsay II

The STEPPS system was used to analyze a Hearsay II phenomenon discovered by Fennell [Fennell 75a, 75b]. He appended a multiprocess simulator to a version of the developing HSII system and measured the processing performance under several multiprocessor configurations. One of the parameters of interest to him was the effect of locking on the the throughput of the multiprocessing system. Throughput is important to the speed with which the HSII system would perform the speech understanding task. Measures of throughput that he used were:

1. The average number of active processors, and

2. The average number of inactive processors.

One of Fennell's results was that when locking was used, to insure data integrity and to prevent deadlocks, he obtained a measure of throughput averaging 4.16 processors with 16 processors available. However when the locking structure of the simulation was turned off,† the average number of active processors was found to be 11.84. Fennell did not explain this phenomenon, but noted that the locking interference had a significant effect on effective parallelism [Fennell 75a, 75b].

---

†The removal of the locking, as reported by V. Lesser of the HSII researchers, does not affect data integrity since the locking used in Fennell's simulations concerned independent fields of nodes.

The STEPPS system was proposed as a tool to analyze this phenomenon. The motivation was twofold. First, the locking/no locking problem indication of close to threefold processing utilization deterioration was important enough to analyze. Second, this problem appeared to be a practical application of some of the STEPPS facilities.† A factor that added to the appropriateness of the STEPPS model was that it is easy to model a data driven organizational structure, like HSII. One issue for investigation was whether the probabilistic approach to modelling interprocess communications was sufficiently powerful to reproduce the phenomenon found using Fennell's simulation. If successful, the STEPPS model could be modified for representing costly HSII system modifications, and predictions could be made of their effects on HSII performance.

A brief discussion of a pertinent part of the locking algorithm follows (See [Fennell 75a] for complete details). The data base consists of a set of nodes arranged in a two-dimensional structure. Along one dimension are 9 categories called ***lexical levels***. The second dimension represents ***utterance time*** and is divided into 30 distinct units. Thus a node exists in a lexical level at a given utterance time. Nodes can be grouped into ***time regions*** covering all nodes on a single lexicon level occurring between time a and time b ($a \leq b$). Locks can be performed on individual nodes or on regions--locking all nodes within the regions.

In order to prevent deadlocks, locking is performed in a hierarchical manner using specified conventions. The hierarchy is that locks occur in the order: by lexical level and then by increasing time. Each process performs all of its locks, performs some processing, and then releases all of its locks. There can be no deadlocks since all required data nodes must be available before any processing occurs and all nodes

---

†It was not originally recognized that some limitations of the STEPPS system would also be identified. This will be discussed later.

are released before any new locks are performed. In addition, when two processes attempt to lock the same pair of nodes (possibly among other nodes as well), they can not mutually block each other since they both must perform their locks in the same order.

An additional attribute of the HSII locking convention is that a process maintains a lock on a node until it releases all of its nodes. This means that if a process locks node A but is blocked from locking node B, it waits for the release of node B before continuing and maintains its lock on node A while being blocked. This method guarantees that each process will eventually complete its required processing, but the method can cause a third process to be blocked unnecessarily if it only tries to lock node A.

III.C.5. The STEPPS simulation of the locking problem

An analysis of the Hearsay II knowledge sources and preconditions was performed to determine the parts of the blackboard examined by each process type. By executing the HSII prototype system in a sequential mode with data collection features turned on, members of the HSII development team† generated data that was analyzed to determine proper probabilities and computation times used in the STEPPS model of HSII.

Due to the STEPPS system overhead, the complete set of possible locking structures could not be modeled. Thus the STEPPS model of HSII approximated the locking structure. For the simulation of HSII it was determined that locking occurred in only 23 ways with respect to lexicon level. Figure III-32 shows the matrix representing these locks and which processes performed the locks. Each process

---

†Special appreciation is acknowledged to V. Lesser, R. Fennell, and G. Gill.

could choose from among its possible locks (as shown in the table) uniformly as the current locking set for the process.

Lock No.	Lexicon Levels Locked	Lock No.	Lexicon Levels Locked
1.	WORD + WRDSURN	13.	PHON + PSEG
2.	WORD + WRDSURN + SURN	14.	PSEG + SEG
3.	WORD + SURN + PHON + MXN	15.	SHDSENT
4.	WRDSURN + SURN + PHON	16.	SHDWORD
5.	WRDSURN + SURN	17.	WORD
6.	SURN + PHON + MXN + PSEG	18.	WRDSURN
7.	SURN + PHON + MXN	19.	SURN
8.	SURN + PHON	20.	PHON
9.	PHON + MXN + PSEG	21.	MXN
10.	PHON + MXN	22.	PSEG
11.	MXN + PSEG + SEG	23.	SEG
12.	MXN + PSEG		

Process\Lock	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
PRE|RPOL																x	x	x	x	x	x	x	x
KS|UV	x	x		x	x		x	x	x	x	x	x		x									
PRE|PSYN																						x	
KS|PSYN								x	x		x	x										x	x
KS|CSEG								x	x		x	x										x	x
PRE|PSC			x			x																	
KS|SEARCH		x	x	x	x	x	x	x				x											
KS|TIME		x	x	x	x	x	x	x				x											
PRE|UTTB				x	x																		
KS|UTTB			x	x	x	x	x					x											
PRE|SEG																						x	x
KS|SEG																							x
PRE|ALO										x			x										
KS|ALO	x	x			x										x			x					

Figure III-32. Hearsay II Locking Structure Matrix

The thrust of the simulation experiment was to reproduce Fennell's results using a probabilistic model. Appendix B contains definitions of the probabilities used for the simulation. The first question of interest was how the simulation performed with locking vs. without locking.

The individual time divisions for locks also contribute to locking interference. A second interesting question was how the region sizes affected simulated interference. The STEPPS system posed an overhead limitation on what could be modeled and so hindered answering this question. Specifically, it was not possible to represent locking in all of the 30 possible divisions (4680 possible regions). Instead each lexicon level was considered as a single region and decomposed into subregions in successive experiments until the overhead of running the STEPPS system overwhelmed the computer.†

The parameters that could easily be altered for the system simulations were:

the existence of locking,

the number of subregions for each region,

the number of processors available, and

the probabilities that the processes performed their locks.

The region locks for each process were formed by examining the program structures for each of the modeled processes. The probabilities used by a process to choose locking structures were assigned uniformly over the possible locks. The times between locks and the time for a lock to take place were taken from the HSII system data.

Several models of the system were simulated and representative results are shown in Figure III-33. The results demonstrate that with no process interference there can be 12.26 processors active on the average. This corresponds to the results found by Fennell's simulation of the entire HSII system. The second set of results (with locking) shows that when the region locking interference is introduced there is a dramatic decrease in parallel processing. As the regions were further decomposed, parallel processing did not substantially change.

---

†We ran out of memory at 200,000 words on the PDP-10.

Locking Strategy	Avg. Active	Subregions	Total Locks
No locking interference	12.26	9	23
With locking	3.11	9	23
Subregions MXN(2), PSEG(2)	3.06	11	53
Subregions MXN(2), PSEG(2), PHON(2)	3.27	11	75
Subregions MXN(2), PSEG(3), PHON(2)	3.11	13	85

Figure III-33. Hearsay II Representative Results

As discussed in Appendix C, the statistical validation of these results, based on the elimination of initial condition bias, was accomplished by performing trial runs of the Hearsay II model to deterime subsequent simulation experiment run times. Confidence intervals were not determined for the statistics presented since accumulated statistics (i.e., average active processors) requires multiple simulations [Gordon 69] which were felt to be too expensive. Moreover, the STEPPS Hearsay II simulation results were correspondences to Fennell's simulation experiments, which were also not validated [Fennell 75a].

The STEPPS simulation results demonstrate that the probabilistic approach can be used to model the Hearsay II multiprocess communication structure. Both the Fennell and the STEPPS simulations indicated about a threefold decrease in a measure of processing throughput due to locking. In addition, the relatively simple STEPPS model indicated that the granular locking structure used by Hearsay II may not be necessary.

III.C.6. Reflections on the STEPPS Hearsay II simulation

The STEPPS system's use as a tool for examining the Hearsay II process structure was successful in that STEPPS adequately represented major interprocess communication dependencies and produced results reflecting on the Hearsay II system

structure. The probabilistic approach applied within the STEPPS structure and the approximations to the actual implementation were sufficiently powerful to reproduce Fennell's result and indicate an area for HSII system modification. Another significant observation was that the data used to reproduce the Fennell result came from a sequential operation of HSII and yet yielded appropriate predictions concerning the multiprocess HSII system. This observation implies that the HSII multiprocess structure does not produce a large amount of interprocess assistance (or interference) over the STEPPS multiprocess model that contains no direct interprocess assistance.

Some further simulation experiments might have been useful for studying Hearsay II. However, during the STEPPS simulations the Hearsay II system process structure was altered. These modifications included the replacement of several Precondition and Knowledge Source processes with new versions which resulted in an increase in the total number of processes. To incorporate the Hearsay modifications would have required the collection and analysis of data from Hearsay and the creation of a new STEPPS model. The cost in computer time and analysis effort was too large during the period that the simulations were performed. Experiments that might have been useful are:

> Restrict the number of available processors instead of using the maximum possible.
>
> Modify the process structure to use many simple Precondition and Knowledge Sources.
>
> Increase the number of subregion locking beyond that used.

An additional limitation to performing these simulation experiments was the STEPPS system itself, since prototype limits of the STEPPS system were reached when the Hearsay II simulation model exceeded available PDP-10 memory.

Even considering the previously discussed limitations, the STEPPS system

application to Hearsay II was significant. First, the STEPPS model could easily represent the non-trivial HSII communications structure. Part of this ease was due to the HSII data directed process organization of interest in the experiment being well suited to the probabilistic nature of STEPPS processes. The application demonstrated that the data collected during a STEPPS simulation† was sufficient to provide the required results.‡ Finally, the STEPPS system could really aid the HSII systems developers in tuning their system by providing a relatively simple framework to examine the consequences of parameter changes (e.g. probabilities and timing) in addition to structural changes.

---

†See Chapter V for details on simulation data collection and parameters.

‡This can also be stated for the Bliss/11 application.

Chapter IV

Analysis of a STEPPS Model

A STEPPS model of a program can be analyzed to predict some of the program's performance properties. Unless a model is analyzed and certified as safe, a program that is constructed, based on the model, may be useless. It is sometimes valuable to exploit the similarity of the STEPPS model to known models for application of known analysis techniques; thus we begin with a review of these models and techniques.

IV.A. Markov and semi-Markov processes

The model of a process described in Chapters I and II is essentially a description of a semi-Markov process [Howard 71 vol. 1 & 2]. A discrete-time ***Markov process*** is a probabilistic system composed of a set of states, a designated current state, and a probabilistic rule for changing between states. The basic rule for a Markov process is that the probability of a transition between the current state and any successor state is independent of any past history. Let $\{E_i\}^n_{i=1}$ be the set of successive events and let the finite set $\{X_j\}^m_{j=1}$ be the possible state values[†]. Then the Markov assumption is formally:

$$P(E_{n+1} = X_k \mid E_t = X_{j_t},\ t = 1, \ldots, n) = P(E_{n+1} = X_k \mid E_n = X_{j_n}).$$

The probability that the next event, E_{n+1}, is a particular state, X_k, is only dependent on the last event X_{j_n}. When finite state processes are studied, the probabilities are

---

[†]In general the state values could be an infinite set, but this research is only concerned with finite state processes.

sometimes chosen to approximate known distributions to facilitate analysis. In all cases, the sum of the probabilities of transferring from a particular state to the set of next possibilities must be 1.

A Markov process may be composed of *chains* of states. A chain is a set of states such that once the process enters one of the states of the set the only other states that the process can enter are in that set. In general, a process may have more than one chain and whichever chain is entered first determines how the process will eventually perform. The analysis and operation of a process with more than one chain is dependent on the process's initial state. A process with only one chain is called a ***monodesmic process.***

For a monodesmic Markov process it is still possible that some states do not recur. This happens if the process can ever reach a state such that the probability of ever reaching some states is zero. States that can not recur in *steady state*† are called *transient states.* Informally, a transient state is a state of a process that can only be entered between an initial state and a chain.

Example IV.A-1

Figure IV-1 (a) shows the transition matrix of a Markov process with two chains. The states of the process are w, x, y, and z. If the process is initially in either state w or x then the only states that it can ever enter are w and x. However if the process is initially in either state y or z then it can only enter states y or z. Thus the process has two chains. No states are considered to be transient since all of the states are in some chain.

Figure IV-1 (b) shows the transition matrix of a monodesmic process having two transient states. The states of the process are a, b, c, and d. The chain is composed of states c and d since once they are entered no state other than either of them may be entered. In addition states a and b do not form a chain since the process may eventually enter the c - d chain from a and b. If either a or b is an initial state they may recur many times, but eventually the chain will be entered and then it will be impossible to enter either of them again.

---

†*Steady state* is defined to be the operation of the process after some suitably large number of transitions.

	w	x	y	z
w	p	1-p	0	0
x	q	1-q	0	0
y	0	0	r	1-r
z	0	0	1	0

(a) Two chains: (w,x) (y,z)

	a	b	c	d
a	0	p	1-p	0
b	q	0	0	1-q
c	0	0	0	1
d	0	0	1	0

(b) Transient States: a and b

Figure IV-1. Markov Processes

Markov processes have been studied in order to solve problems such as:

> What is the expected number of transitions before entering state S?
>
> What is the probability of entering state S from state T: (1) in m transitions? (2) in m or fewer transitions? (3) ever?
>
> In steady state, what is the probability of entering state S on the next transition?

The last question points out one example where steady state activity is considered important. For monodesmic processes the initial state is unimportant, but the activity of processes with multiple chains is strongly dependent on the initial state since as shown in Example IV.A-1 a process can behave quite differently in steady state depending on how it was initialized. For this reason most models using Markov processes are monodesmic.

This research is also concerned with the steady state properties of a multiprocessing program. Transient states create difficulties in analyzing data flow in the steady state of a multiprogramming model because it is possible that a process will

never reenter a transient state. The STEPPS model is restricted to disallow processes with multiple chains and transient states because they do not contribute to the steady state of a process. The STEPPS system is able to analyze a process and determine whether these restrictions have been met. The algorithms for performing this analysis are discussed later in this chapter.

A *semi-Markov process* is a generalization of the Markov process model. In a Markov model, one unit of time elapses between successive transitions in all cases. In the semi-Markov model, the time taken between successive transitions depends on the particular transition. In the model's most general form, the time taken between any two successive states can be a random variable; in the STEPPS model this serves no useful purpose, so the time taken between any two particular transitions is a constant depending only on the two states. In fact, the real time between transitions in a STEPPS model is usually not completely predictable since a process may be forced to wait as discussed in Chapters I and II.

Some problems that have been studied using the semi-Markov process models are:

What is the expected process time between entering state S and entering state T?

What is the expected process time between recurrences of state S?

In steady state, what is the expected percent of time spent in state S?

Again, the last question is the most interesting one for the STEPPS model. There is not always an accurate result for a STEPPS model because processes in the STEPPS model are not semi-Markov due to the essentially unpredictable† wait time. However an estimate of the type of activity that a process will be performing when it is executing is still a useful result.

---

†The wait time is unpredictable for a given process when considering the process independently of the entire model.

The theory tells us that it is possible to predict the steady state probabilities of which state will be entered next, not knowing the present state. This means that it is possible to create a representative transition matrix such that each row is the same, i.e. the choice functions are all the same when the most recent state is unknown. These probabilities also reflect the probability of being in each of the states after a large number of transitions.

The steady state probabilities can be determined analytically by solving a set of n+1 linear equations in n unknowns. Let ST_i, i=1, . . . , n be the steady state probabilities of the process and let $p_{i,j}$ be the probability of entering state j from state i in one transition. The equations to be solved are:

$$ST_i = p_{1,i} * ST_1 + \ldots + p_{n,i} * ST_n \qquad \text{for } i = 1, \ldots, n$$

$$1 = ST_1 + ST_2 + \ldots + ST_n$$

The first n equations are redundant, so the solution requires replacing one of the first n equations with the last equation. The system of equations will be solvable since the matrix describes a monodesmic process with no transient states† [Howard 71 vol. 1]. Otherwise the equations do not have a unique solution.

The analysis that has just been described is one of the Markov theoretical analytic techniques that can be applied to the processes of a STEPPS model. The fact that the STEPPS model processes are similar to semi-Markov processes is only useful if a system designer wants to analyze components of a STEPPS model in this way. In most cases, Markov and semi-Markov analysis of STEPPS processes is of limited usefulness since the STEPPS processes are only components of a larger model and the transition matrices do not entirely reflect the operation of a process.

In order to represent analytically an entire STEPPS model, all possible states

---

†This is guaranteed by the STEPPS system.

(In the Markov process sense, rather than STEPPS) must be included in the analytic description. Not only must every STEPPS state be included as *analytic* states, but also analytic states must be introduced to represent the operations of the STEPPS links. The effect is the creation of a matrix representing at least N squared states (where N is the sum of the number of STEPPS states in each STEPPS process). Not only is this model complex, it requires the introduction of probabilities (and associated times) for some new, potential Markov state changes.

IV.B. Well-formed STEPPS models

As noted in Chapter I, in order for a STEPPS model to be useful it must meet certain restrictions and be designated as a *well-formed model*. Earlier in this chapter it has been pointed out that each process in a STEPPS model must be monodesmic and have no transient states (termed *well-formed process*). An additional restriction guarantees that a model represents a data flow which can be simulated and which can reach steady state if simulated for a sufficient period of time. Hence other restrictions to the model (termed *well-formed graph*) are that all links must be attached to both input and output ports, that all ports be attached to links, and that the graph be connected. If these restrictions were not imposed then some process would eventually request messages from an empty link or try to send messages to a link whose message limit has been reached.

IV.B.1. Monodesmic and transient state well-formed criteria

Let the N states of a process be identified by the integers 1 to N. Let the probability of state j succeeding state i be $p_{i,j}$, the entry in an N by N transition matrix P.

In order to test whether a process is monodesmic and has no transient states it is sufficient to determine whether the probability of each state transferring to each other state in N or fewer transitions is greater than zero. The first step is to form a ***transition relation matrix***, C, representing a relation between states, defined by $C_{i,j}$ = 0 if $p_{i,j} = 0$; and 1 otherwise. The next step is to form the transitive closure of the C matrix, which describes whether there exists some succession of connections between any two states. If the closure of C is all ones then every state is able to transfer to every other state and so the corresponding process is monodesmic and has no transient states.

There are several algorithms for forming the transitive closure of a relation. One method to form the closure, as shown by Prosser [Prosser 59], starts by forming Boolean powers of the matrix to show whether a transition can occur in two or more transitions. The i,j term of the Boolean square of a matrix is the Boolean expression (logical sum):

$$(C^2)_{i,j} = \Sigma^N_{k=1} C_{i,k}C_{k,j}$$

The i,j term is equal to 1 if and only if there is some k such that $C_{i,k} = C_{k,j} = 1$. Similarly if the N-th power of the matrix is formed, a 1 in the resulting matrix represents that a transition can be made in N steps. If a Boolean sum is taken of the first N powers of C, then the resulting matrix represents whether a transition can be made in N or fewer steps. This matrix is the closure. (Other, more efficient, methods for forming the closure of a matrix are known [Warshall 62].)

IV.B.2. Well-formed graph structure criteria

Three basic structural properties are necessary for a well-formed STEPPS graph model. They are:

1. Each port is attached to a link.
2. Each link is attached to at least one input port and one output port.
3. All nodes are connected to each other via some set of paths, i.e. the graph is not disjoint.

The last property means that for any partition of the nodes of the graph into non-empty subsets of nodes, there will exist at least one connection from some node in each subset to a node outside of that subset.

The first two properties are verified by examining each node of the graph and checking the connections to the node. The third property is determined by first forming a node connection matrix NC where $nc_{i,j} = 1$ if node i is connected to node j or if node j is connected to node i; 0 otherwise (ignoring that the graph is directed). As before, the closure of the NC matrix is formed. If the closure contains all ones, then there exist connections between every pair of nodes.

IV.C. Deadlock structures and situations

The nature of communication dependencies can create problems for a system of interacting communicators. The basic problem in a STEPPS model is that processes can achieve states such that at least one process will never be able to change state because it is waiting to activate its associated port; this is called the ***deadlock problem.*** In some STEPPS structures a deadlock problem may be so severe that no process can ever change state and no further processing of any kind is possible. On other structures some subparts may still be able to continue processing (possibly incorrectly). A structure that is completely deadlock free is defined to be ***safe.***

Either of two views may be taken when examining a structure for deadlocks. The first view is that a structure must not contain any chance of an occurrence of a

deadlock. The second is that a process may have deadlocks as long as it is possible to identify how the deadlocks occur and the probability of their occurrence. The term "deadlock" in the STEPPS model refers only to communication structures which can not be removed other than by restructuring a model. In practice other methods, e.g. as restarting a process after an unusually long delay, are sometimes used in systems where deadlocks can occur.

The deadlock problem has been studied extensively along several dimensions. The survey by R. C. Holt [Holt 72] examines many of the deadlock problems. Most of the deadlock algorithms are oriented toward solving problems concerned with resource requests from a pool of resources. Holt presents a graph model of the resource problem and a set of graph reductions to determine whether a modeled system contains deadlocks. The difference between the STEPPS solution and his is that Holt limits his analysis to necessary conditions (cycles) and sufficient conditions (a knot†) for the existence of a deadlock. He does not report on the solution of the general problem. Several problems that have been solved have been concerned with reusable resources. The STEPPS model does not consist of reusable resources since messages, which are the resources in STEPPS, need not be preserved. The STEPPS model is somewhat different from the models that have been examined in past research, so the deadlock problem has been examined and solved (with a few restrictions) for the STEPPS model.

The following sections present some structures and situations that can cause a STEPPS model to deadlock. These examples are not necessarily independent nor complete, but they demonstrate some types of deadlock structures.

---

†A *knot* is a subset of nodes of a directed graph such that each node is attached to the other.

IV.C.1. Initial condition incompatibility

It is easy to create a structure that is safe for some initial states of the components of the structure, but not for others. The possible problems are that data are not available where required or that the system contains too much data. For example, all output ports that are initial states might be attached to links that are initially at capacity and all initial input ports might be attached to empty links. It is not necessary for such a condition to occur before execution begins; the condition may also occur after only a few state transitions. An example of initial condition deadlock is shown in Figure IV-2. In this example process C is waiting for a message from B and process B is waiting for a message from C. Process A will always be waiting to send a message.

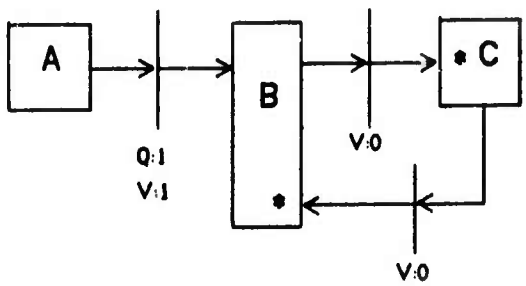


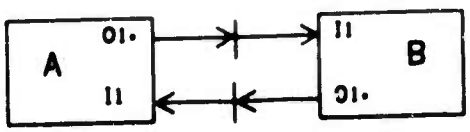
Figure IV-2. Improper initial condition

IV.C.2. Loops

A ***loop*** is defined to be a path from an output port, O_x, of a process to an input port, I_y, of that process with no connections along the path between them going to the original process. When each node in a loop is connected only to other nodes in the loop, the loop is called ***a closed loop.***

If any port of any process node is immediate-recurrent (including the nodes at either end), then it is possible that the port could send (or request) extra messages. A solution to this deadlock problem is that both a SINK† and a SOURCE‡ must be attached to nodes in the loop. Thus the loop cannot be closed when there is an immediate-recurrent state within it (Figure IV-3).

A loop that is not closed and does not have both a SINK and a SOURCE attached to nodes in the loop may contain deadlocks because it may be possible for a message to be shunted to the SINK or any other process not in the loop. Similarly, extra messages entering a loop from a node not in the loop can cause a link to become filled with extra messages.



A	11	01
11*	0	1
01	1	0

B	11	01
11	p	1-p
01*	1	0

Figure IV-3. Loop with immediate-recurrent states

---

†A *SINK* is process whose only port is an input port.

‡A *SOURCE* is a process whose only port is an output port.

IV.C.3. Incompatible sequences

When a data path can be recognized as a *closed path,*† it is possible to determine the number, N, of messages required to enter this path in order for any messages to be available at the link attached to the end of the path. It is also possible to determine the number, M, of messages that will be available at the end of the path. The link attached to the end of the path may require a certain number of messages, L, before the input to the next path attached to it can yield any messages. If M does not at least equal L and if N, M, and L are finite then the system can deadlock. It also must be true that fewer than 2N messages enter the path before a response is required from the path. Figure IV-4 shows an example of this.

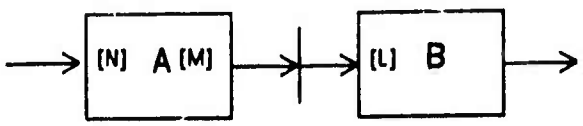


Figure IV-4. Incompatible Sequence

IV.C.4. Split paths that do not join properly

A data path may split in two ways. If a link is attached to more than one input port, messages that reach the link may go down either path. If the paths join again at two different ports of the same process then it may be possible for an insufficient number of messages to enter one of the paths and thus force the merging process to wait for data that will never come. Figure IV-5 shows this situation, where processes A and B send messages to C and C must receive a message from one before receiving a message from the other.

---

†A *closed path* is a path between two nodes such that all nodes in the path are attached only to other nodes in the path.

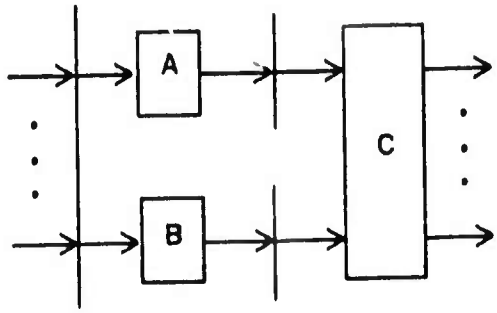


Figure IV-5. Link split paths

The second way in which data may go down alternate paths occurs when a process sends data along two different paths that eventually merge. If more data can go down one path than can be received by the port at the end of the path then this path will eventually fill up with messages. The two processes must be exactly synchronized as to their data dependencies. Figure IV-6 shows this situation. Every message sent by A from port A.01 must be accepted by B.I1 and the same is true for A.02 and B.I2.

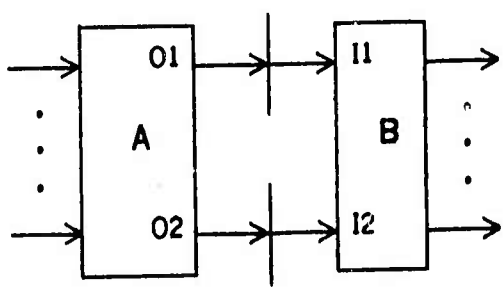


Figure IV-6. Process split paths.

IV.D. Reducing a STEPPS model

Under certain conditions it is possible to determine whether a STEPPS graph

model is ***deadlock-free***. The conditions are that the graph be well-formed, initial process states be ignored, and the initial number of messages and queue size limits be ignored. The method used to determine whether a graph is safe is to apply a set of ***graph reductions***. These reductions will be shown to reduce all safe graphs to other safe graphs and to reduce all unsafe graphs to other unsafe graphs. In addition, it will be shown that one of the reductions is always applicable to a safe graph. Thus, the reduction process may be repeatedly applied until either an empty graph or an irreducible graph is reached. When an empty graph is produced the original graph is safe. When an irreducible graph results, then the original graph can generate a deadlock.

There are four graph reductions that can be applicable when certain conditions are met:

R1: Combine two adjacent processes.

R2: Eliminate states of a single process.

R2a: Combine two ports of the same type, attached to the same link, to become one port.

R2b: Eliminate ports of opposite type connected to the same link.

R2c: Eliminate ports attached to SOURCE/SINKS.

R3: Combine two processes that are ***in-parallel***†.

R4: Eliminate all SOURCES, SINKS, and unattached links.

Graph Reduction Process: The first three reductions are applied iteratively until none is applicable and then the last, R4, is applied. If the result is an empty graph then the model is safe; otherwise the model is unsafe. The reduction process sometimes converts the graph into disjoint parts, and this is necessary to the reduction process.

---

†Two processes are ***in-parallel*** when each process has exactly one input port and one output port and the input ports of the respective processes are connected to the same link and the output ports are connected to the same link.

The reductions are based on potential interprocess communications. Since a process *transition relation matrix* represents the presence or absence of possible interprocess port activations, it will be the vehicle used to demonstrate that the reductions maintain process legality. Thus by proving that the transitive closure of a resultant transition relation matrix is entirely 1's, each reduction is demonstrated as producing resultant processes that are monodesmic and have no transient states.†

IV.D.1. R1: Combine adjacent processes

Two *adjacent processes*‡ are combined when it is determined that their data manipulation functions can be replaced by a single process. It will be demonstrated that the combination of two adjacent processes in an unsafe graph will not convert the graph into a safe one.

R1 is applicable in two situations:

R1a: neither of the adjacent ports is immediate-recurrent and they repeat the same number of times.

R1b: one of the processes is a DELAY††.

For R1a, the two processes are combined and the link between them is eliminated. For R1b, the DELAY and the link between the processes are eliminated. R1b is a trivial case where the DELAY is functioning as a link. The remainder of this subsection is concerned with R1a.

R1a relies on the assumption that each of the two adjacent processes will

---

†Some representative probabilities can be assigned to the resultant processes, but these will not be presented since they detract from the clarity of the explanation of the reductions.

‡Two processes are *adjacent* when they contain adjacent ports.

††A *DELAY* is a process with only two ports, an input port and an output port, provided neither port is immediate-recurrent.

eventually enter the states of their adjacent ports. The situation can be modeled as one where the process containing the output port sends a message to the other process (and waits), and this second process computes until it requires another message from the first process. Then the first process computes until it reenters the original state. In this way, both processes are able to change state if one can (when the graph is safe). The transition relation matrix of the combined process is formed by a construction that (i) eliminates the adjacent ports and (ii) unites the successors† of states that immediately preceded an eliminated state of one process with the successors of the state of the second process. In this way the new combined process is still monodesmic and without transient states since each state is still able to enter each other state, but now may go through states of what was formerly part of a different process.

The new transition relation matrix is formed in the following manner. Let the ports A.e and B.f be adjacent and let neither state be immediate-recurrent. Let A.x, A.z, B.y and B.w be other ports of the two processes. If the new combined process is called AB, then c'(AB.x,AB.z) = c(A.x,A.z), c'(AB.y,AB.w) = c(B.y,B.w), c'(AB.x,AB.y) = c(A.x,A.e) ∧ c(B.f,B.y) and c'(AB.y,AB.x) = c(B.y,B.f) ∧ c(A.e,A.x)‡.

Lemma R1.1: If A.e succeeds A.x and B.y succeeds B.f, then AB.y succeeds AB.x, i.e. c'(AB.x,AB.y) = 1.

Proof: A.e succeeds A.x means c(A.x,A.e) = 1 and B.y succeeds B.f means c(B.f,B.y) = 1. Therefore c'(AB.x,AB.y) = c(A.x,A.e) ∧ c(B.f,B.y) = 1 ∧ 1 = 1.

Lemma R1.2: If B.f succeeds B.y and A.x succeeds A.e, then AB.x succeeds AB.y.

---

† The *successor states* of a state are those that can be entered in one transition.

‡c(s,t), a transition relation matrix entry, is defined to be the presence (1) or absence (0) of probability of entering state t from state s. c'(w,u) is a transition relation matrix after the application of a reduction. All operators (∧ and ∨) are logical operators.

Proof: As in Lemma R1.1.

Lemma R1.3: There exists a sequence of transitions from AB.x to AB.y.

Proof: Since A and B are assumed to be legal STEPPS processes, their respective transitive closure transition relation matrices are all 1's. As a property of transitive closure, this means that there exists a sequence of transitions from A.x to each predecessor of A.e and their respective similar states in AB. Similarly there exists a sequence of transitions from successors of B.f to B.y and their respective similar states in process AB. By Lemma R1.1 and the above, there exists a sequence of transitions from AB.x to each corresponding successor of B.f and thus to AB.y.

Lemma R1.4: There exists a sequence of transitions from AB.y to AB.x.

Proof: As in Lemma R1.3.

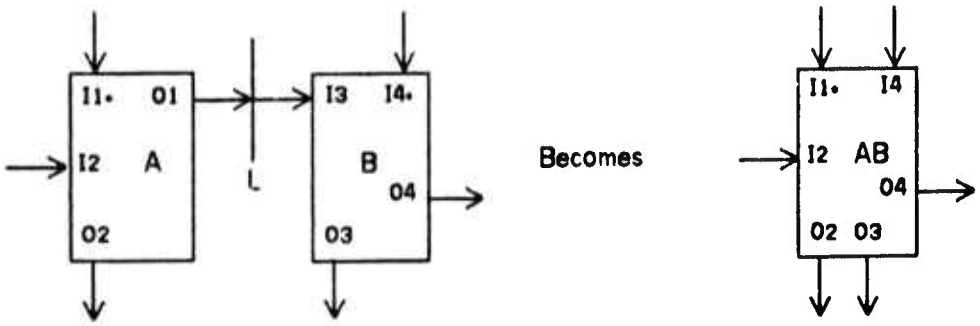
Lemma R1.5: Process AB is a legal STEPPS process, i.e. the transitive closure of the corresponding transition relation matrix is all 1's.

Proof: By Lemmas R1.3 and R1.4, there are sequences of transitions between each state that was originally in A to each state originally in B and visa-versa. Therefore by juxtaposing sequences of transitions there exist sequences of transitions between any two chosen states of AB. By the definition of transitive closure this corresponds to all 1's in the transitive closure transition relation matrix for AB.

Example IV.D-1

> The processes in Figure IV-7 are combined by forming a process with two states fewer than the total number of states of the original two processes. As shown by the transition relation matrices, all of the predecessor states of the output process now transfer to the successor states of the input process. All of the predecessor states of the input process now transfer to the successor states of the output processes.

Theorem R1: R1 (combine adjacent processes) preserves the message flow structure of a model with respect to graph elements not involved in the reduction (whether or not the original graph was safe).



A	I1	I2	O1	O2
I1	0	0	0	1
I2	1	0	1	1
O1	0	1	0	0
O2	1	1	1	0

B	I3	I4	O3	O4
I3	0	1	1	1
I4	0	0	0	1
O3	1	0	0	0
O4	1	1	1	1

AB	I1	I2	I4	O2	O3	O4
I1	0	0	0	1	0	0
I2	1	0	1	1	1	1
I4	0	0	0	0	0	1
O2	1	1	1	0	1	1
O3	0	1	0	0	0	0
O4	0	1	1	0	1	1

Figure IV-7. Process combinations

Proof: Let A and B be the original processes and let AB be the result of combining them. It will be shown that any message that could be requested by A or B can be requested by AB and that any message that would be sent by A or B to a link will be sent by AB.

By Lemma R1.5, the new process is monodesmic and has no transient states. Coupled with the reduction definition, this means that all states of AB can be entered exactly as often as in the original processes, A and B. Thus all input ports of AB are guaranteed to be able to receive messages if they originally could, so the state

associated with a given input port will always be able to change. Likewise each state associated with an output port of AB can change to another state if it could originally. For these two reasons all messages that would be requested by A or B will be requested by AB and all messages that would be sent by A or B will be sent by AB.

IV.D.2. R2: Eliminate states of a process

There are three circumstances in which a state of a process may be removed by applying reduction R2. There are two distinct methods of removing a state: combine two states to become one; and eliminate a state. As with R1, the removal of a state does not affect data flow patterns. (An exception is that the combination of two states into one sometimes modifies the number of times a state repeats.)

The method used to *combine two states* into one state is defined as follows. Let A.x and A.y be the names of the states of process A being combined. For convenience, the resultant combined state will be called A.x. The rule for combining the states, in terms of the transition relation matrix for process A, is:

Let A.z be a state of process A that is neither A.x nor A.y, i.e. it will remain after the reduction.

$$c'(A.x,A.z) = c(A.x,A.z) \vee c(A.y,A.z)$$
$$c'(A.z,A.x) = c(A.z,A.x) \vee c(A.z,A.y)$$
$$c'(A.x,A.x) = c(A.x,A.x) \vee c(A.y,A.x) \vee c(A.y,A.y) \vee c(A.x,A.y)$$

The above means that any successor of A.y becomes a successor of A.x, and any predecessor of A.y becomes a predecessor of A.x.

Lemma R2.1: The reduction to *combine states* yields a legal process.

Proof: It must be shown that the resultant process has no transient states and is monodesmic. The original process, A, was legal and thus there existed finite sequences of transitions from each state to each other state. The construction of the

new process by ***combine states*** guarantees a legal process since (a) if there existed a sequence of transitions between two states without going through A.y, the reduction does not alter the sequence and (b) any sequence of transitions that went through A.y will now go through A.x instead.

The method used to ***eliminate a state*** of a process is defined as follows. Let A.x be the state being eliminated and let A.y and A.z be other states. The rule for eliminating a state, in terms of the transition relation matrix for process A, is:

$$c'(A.y,A.z) \equiv c(A.y,A.z) \vee (c(A.y,A.x) \wedge c(a.x,A.z))$$

The above means that A.y preceeds A.z either if it did before the reduction or if A.y preceeded A.x and A.x preceeded A.z.

Lemma R2.2: The reduction to ***eliminate a state*** yields a legal process.

Proof: A sequence of transitions between two states not going through A.x still exists after the reduction. A sequence of transitions that went through A.x, simply skips A.x after the reduction. Thus the reduction yields a process that is monodesmic and has no transient states.

R2a: When two ports of the same type are connected to the same link one port is removed, depending on one of the following conditions.

(i) Each of the two states can succeed the other in one transition. This means that the states are equivalent to one immediate-recurrent state. The two states are ***combined*** to become one state.

(ii) The successor states of the two states are the same (not counting each other). This means that the states act as one state with possibly different transition probabilities from the original states. The two states are ***combined*** to become one state.

(Iii) The two states are in-sequence, i.e. one state will enter the other with certainty. Alternatively, they may be one-to-one. This means that the two states are really one with finite repetition. One of the states is ***eliminated***.

Note that a link is also eliminated by reduction R2 when all ports that had been attached to the link are deleted.

R2b: When a link is only attached to both input and output ports of processes, then pairs of these input/output ports of the same process that are one-to-one and repeat the same number of times can be eliminated. When this structure occurs, every message sent to the link is guaranteed to be requested by one of the other ports of the process. If the ports are the last two connected to the link then the link is also removed.

Example IV.D-2

In Figure IV-8, ports I1 and O3 are adjacent and are one-to-one. They are eliminated as shown.

ALPHA	I1	I2	O3	O4
I1	0	1	0	1
I2	0	0	1	1
O3	1	0	0	0
O4	0	1	1	1

Becomes

ALPHA	I2	O4
I2	1	1
O4	1	1

Figure IV-8. Adjacent ports of a process

R2c: A state that is attached to a SOURCE/SINK† is eliminated, since once it is entered, the process can always be assured of being able to enter a new state. If the

---

†A *SOURCE/SINK* is either a SOURCE or a SINK depending on the context. A SOURCE would be attached to an input port, whereas a SINK would be attached to an output port.

state is the last state of a process then the entire process is eliminated. If the port was the last port attached to a link then the link is also eliminated.

Example IV.D-3

In Figure IV-9, both ports I4 and O6 are attached to SOURCE/SINKS. They are both eliminated.

Theorem R2: R2 (eliminate states of a process) preserves the message flow structure of a model with respect to graph elements not involved in the reduction except for links attached to SOURCE/SINKS

Proof: Let A be a process that is reduced to A'. By Lemmas R2.1 and R2.2 each state of A' can always be entered. The cases to be considered are enumerated by looking at how a link was attached to A and then to A'.

A link that was attached to A and not to a port of A that was eliminated by the reduction will still have the same interaction with A' as with A since, by construction, any states that would have entered an eliminated state will transfer to a successor of the eliminated state. Thus the state that is attached to the link will occur just as often in A' as in A.

A link that was attached to A and is attached to A', but by one fewer port, will still have the same interactions with A' as with A since the remaining connections to A' are constructed to guarantee this. Two states of the same type that are attached to the same link and succeed each other act like an immediate-recurrent state since any number of link interactions can occur before a different state is entered. Two states of the same type have the same successor states and are acting in the same manner as one state except for different probabilities to the successor states. Two states of the same type that are in-sequence and are attached to the same link act like one state that repeats before entering another state.

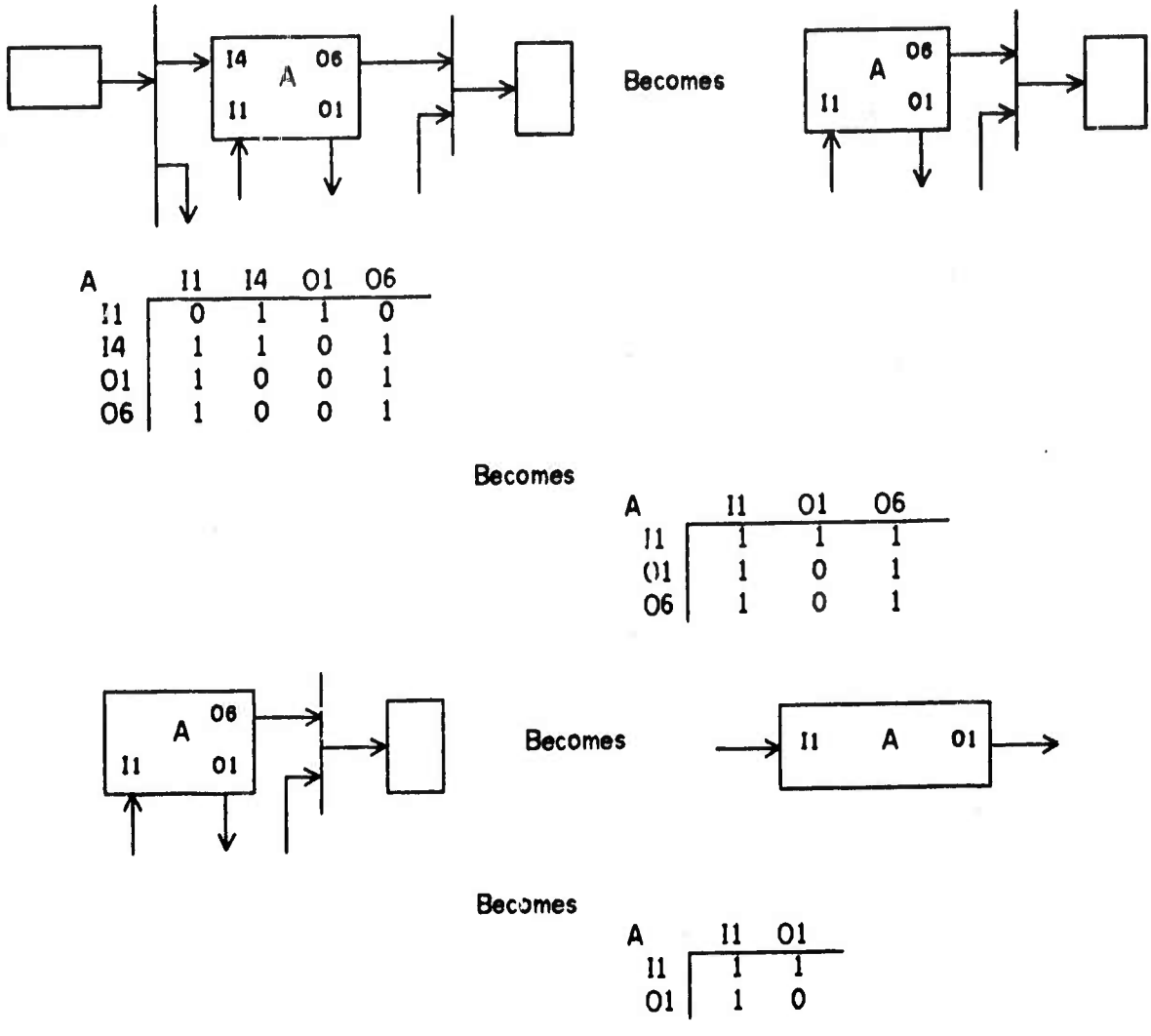


Figure IV-9. Ports attached to SOURCE/SINKS

A link that was attached to A and is attached to A' by two fewer ports occurs when pairs of input/output ports are removed. The message flow is preserved since the ports were only removed if they were one-to-one. This means that whenever a message is sent to (requested from) the link, it is guaranteed that a message will later be requested from (sent to) the link. A link that was only attached to those two ports is removed as part of the reduction. Since the message flows to and from the link

were eliminated with the link, the remainder of the graph is the same. This completes the proof.

By assumption, Theorem R2 is not concerned with links that had been attached to SOURCE/SINKS and are no longer attached to a port of a process. This situation is represented by reduction R2c. Messages flow between a SOURCE/SINK and the reduced process. The reduction occurs by considering the SOURCE/SINKS as message suppliers and terminators. Reduction R4 eliminates these processes and so message flow involving them is eliminated.

IV.D.3. R3: Combine processes that are in-parallel

When two processes are in-parallel, each process has only one input and one output port and both processes' input ports are attached to the same link and both output ports are attached to the same link. When a message is in the queue of the common link attached to the processes' input ports, it can be requested by either of the processes. Whenever the choice will not affect message flow the two processes are combined. In particular, an immediate-recurrent state subsumes the function of the state of the process that is attached to the same link. Thus a DELAY that is in-parallel with other processes containing two states is eliminated.

A *BLACK BOX* is a process having just two ports, one output port and one input port. Both associated states are immediate-recurrent. Any process that is in-parallel with a BLACK BOX can be removed since the BLACK BOX subsumes the operation of the other process.

Let the two processes be ALPHA and BETA with ports ALPHA.I1, ALPHA.O1, BETA.I1 and BETA.O1 (Figure IV-10). The second process, BETA, will be the combined process. The new transition relation matrix is defined by:

$$c'(\text{BETA.I1},\text{BETA.I1}) = c(\text{BETA.I1},\text{BETA.I1}) \vee c(\text{ALPHA.I1},\text{ALPHA.I1})$$

c'(BETA.I1,BETA.O1) = c(BETA.I1,BETA.O1) ∨ c(ALPHA.I1,ALPHA.O1)
c'(BETA.O1,BETA.I1) = c(BETA.O1,BETA.I1) ∨ c(ALPHA.O1,ALPHA.I1)
c'(BETA.O1,BETA.O1) = c(BETA.O1,BETA.O1) ∨ c(ALPHA.O1,ALPHA.O1)

It is obvious from these simple equations that the new transition matrix is legal.

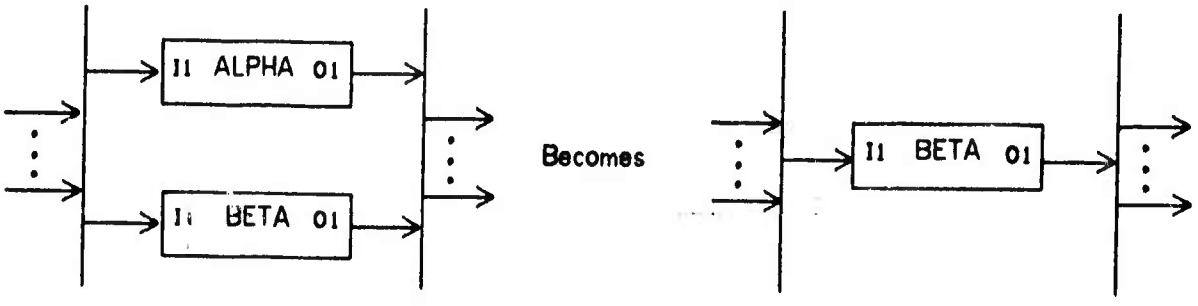


Figure IV-10. Combining processes that are in-parallel.

Theorem R3: R3 (eliminate processes that are in-parallel) preserves the message flow of a model with respect to graph elements not involved in the reduction.

Proof: If one of the input ports that is attached to the link attached to the input ports of the two processes is immediate-recurrent, then it is possible that an undeterminable number of messages can be requested by the processes before a message is sent to the link attached to the processes' output ports. Thus, if one input state is immediate-recurrent then the elimination of the other does not affect the number of messages that can be accepted by the reduction of the pair of processes into one process. Likewise if one of the output ports is immediate-recurrent, any number of messages can be available at the link attached to the output ports and so the other output port is eliminated.

If neither input port is immediate-recurrent then the combination of two input ports is the same as one of them requesting a message twice, so the other can be eliminated. Likewise if neither output port is immediate-recurrent then the combination is the same as one sending two messages to the link and so the other output port can be eliminated.

IV.D.4. R4: Remove SOURCES and SINKS

Since the original graph was well-formed, all links were originally connected to both input and output ports. However, reduction R2c removes all connections of one type to a link. When R2c is no longer applicable, no process is attached to a SOURCE/SINK. Thus SOURCES and SINKS can be attached to links, but serve no other purpose than to have allowed R2c to occur. They are eliminated. If the SOURCE/SINK is the last connection to a link then the link is eliminated too.

Theorem R4: The elimination of SOURCES and SINKS preserves message flow of those elements not attached to the SOURCE/SINKS.

Proof: This is true since reduction R4 occurs after R1, R2, and R3 are no longer applicable and since R2 eliminates all connections to SOURCE/SINKS other than the connections between a link and a SOURCE/SINK. Any other elements in a graph are left unaffected since they are not connected to any SOURCE/SINKS.

IV.D.5. Graph reducibility

The remaining requirements to show the validity of the reductions are that a safe graph is always reducible and that an always reducible graph is safe.

Reducibility Theorem: A non-empty, well-formed, but not necessarily connected, safe graph is always reducible. (Equivalently, an irreducible graph is not safe.)

Proof: Assume the existence of an irreducible graph and consider all possible connections to a link in the graph. The implications of the inapplicability of any of the reductions are as follows. There are four cases:

Case1: A link is connected to only input and output ports of one process.

Since reduction R2 is not applicable, then no pairs of these ports are one-to-one and so at least one of the ports can dominate the activity at the link. This will cause the state associated with the other ports to wait indefinitely since eventually either no messages will be available at the link or the link's finite queue size limit will be reached. This is a deadlock situation.

Case2: A link is connected to only two ports, of the same type, of different processes, i.e. adjacent processes. Since reduction R1 is not applicable, one of the adjacent ports is immediate-recurrent. It is possible for one of the processes to dominate the activity at the link. This will cause the other port to wait indefinitely since eventually either no messages will be available at the link or the link's finite queue size limit will be reached. This is a deadlock situation.

Case3: A link is connected only to ports of the same type of one process connected to the link. Since reduction R2 is not applicable, no pairs of corresponding states (i) succeed each other, (ii) have the same successor states, and (iii) are in-sequence. When a message is requested from (or available to) the link, there is no guarantee which port will request (send) a message first. This makes a difference since the successor states of the two ports are different. There are no SOURCE/SINKS in an irreducible graph so it is impossible to guarantee that another link access will occur due to access from other processes. In addition, there are no DELAYS nor adjacent one-to-one ports of a process and so there are no additional guaranteed link accesses due to the process itself. Therefore a process can deadlock because the wrong port can access the link first.

Case4: At least two ports, of the same type, of different processes are connected to a link. Since reduction R3 is not applicable, none of the corresponding processes are in-parallel. Thus the operation of the model can be affected by

whichever of the processes performs the first link access. It is also possible for one process to dominate the activity at a link. There are no SOURCE/SINKS and so no guarantees of an eventual link access. The situation can cause a deadlock when the wrong process accesses the link first.

It has been shown that all possible connections to the link yield a deadlock. Therefore an irreducible graph is not safe. The contrapositive of this is that a safe graph is reducible.

The arguments of this section have demonstrated that an irreducible graph is unsafe and have proved the Reducibility Theorem.

Irreducibility Theorem: An unsafe graph is not always reducible. (Equivalently, a graph that is always reducible is safe.)

Proof: Let ***X*** be a graph that is always reducible. Assume that ***X*** is not safe. It will be shown that this is impossible.

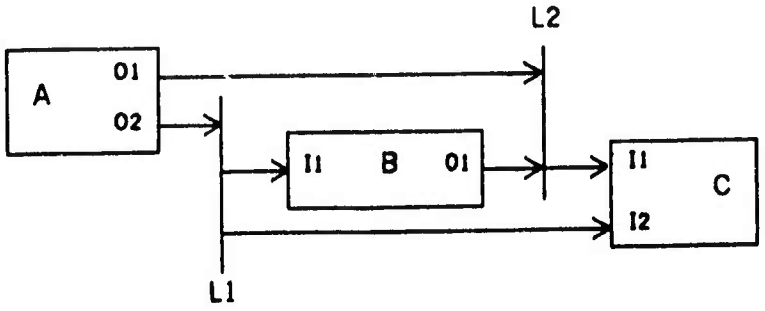
By Theorems R1, R2, R3 and R4, the reductions R1, R2, R3 and R4 each preserve potential message flow in the graph with respect to those graph elements not involved in the reduction. Thus no reduction can cause a deadlock due to interprocess communication not involved with the reduction. Further, by the definition of each reduction and by Lemmas R1.5, R2.1, R2.2 and Theorem R3, a reduction is only applicable to a safe element structure and produces a legal and a safe element structure. Thus a reducible graph is safe.

Example IV.D-4

Figure IV-11 shows an example of an irreducible graph since the following set of state transitions could occur in sequence:

1. A.01
2. C.11
3. A.02
4. B.11
5. B.01
6. A.01

7. Repeat 3 to 6 until the link L2 becomes full.
8. C tries to perform C.I2, but can not since L1 is empty. B can not change state since L1 is empty. A can not change state since L2 is full.



A	01	02
01*	0	1
02	1	0

B	I1	01
I1*	0	1
01	1	0

C	I1	I2
I1*	0	1
I2	1	0

Figure IV-11. An irreducible graph

IV.E. The recognition of deadlocks

Graph Reduction Theorem: Assuming that a STEPPS model is well-formed, that initial conditions are ignored and that queue size limits are ignored, the Graph Reduction Process will yield an empty graph if and only if the original graph is safe.

Proof: By the Reducibility Theorem a safe graph is always further reducible. By the Irreducibility Theorem an always reducible graph is safe. Thus after the Graph Reduction Process is completed, if the result is an empty graph then the original graph was safe; otherwise the original graph was unsafe. This completes the proof.

The graph reductions do not solve the problem of initial state incompatibility. This situation can be recognized by examining each process's initial state and the attached links. If all initial output ports are attached to links that are full and if all initial input ports are attached to links that are empty then the model is initially deadlocked.

It is still possible for a model to enter a deadlock before the model reaches steady state due to link queue length limits and initial queue volumes. A solution to this problem is a requirement that no link attached to an initial output port be full initially. Also these links can not be attached to input ports that are the initial states of another process. In addition, not all of the processes can be in initial states that are connected to links containing no messages. This is not an optimal solution since a system may still be safe if some output ports are initially full. However the requirement is a reasonable one to model and can be altered easily when steady state properties are known.

Chapter V

The STEPPS Simulator and STEPPS Interactive System

Since many performance properties of realistic programs are difficult to treat analytically, a STEPPS model is simulated to collect data which, in turn, is analyzed to predict performance properties of a program. The STEPPS simulator and the implementation of the STEPPS system are presented in this chapter.

V.A. Simulation objectives

An issue concerning the structure of a multiprocess program is whether a particular program decomposition can be improved, i.e. is it a good decomposition. As noted in Chapter I, this research does not address the issue of whether the designed program solves the problem under consideration. The STEPPS simulation facilities serve to enhance predictive performance understanding in the situation where a model is so complex that it is essentially analytically intractable. Another situation occurs when a model may be analytically tractable, but very time consuming to solve, especially when some modifications are made to it.

Specific performance issues concerning a STEPPS model are the following:

1. How much time will be spent computing for each state of a process?
2. In which states will a process be waiting and for how much time?
3. Are the queue sizes too small or too large?
4. What are the expected rates of data flow to a link and of requests from a link?
5. What is the overhead due to interprocess communication?

6. Which sets of processes tend to be active at the same time and which processes are usually active at different times?

7. How many processes are active on the average?

8. What are the effects of limiting the number of available processes and using a variety of scheduling algorithms?

The first question can be answered by performing the semi-Markov analysis described in Chapter IV. However, all of the other questions are more difficult to answer because they deal with interprocess activity.

Consider the question of how long a process will wait in each state. This value may depend upon all of the possible interactions in a model. For example, consider the ring of processes shown in Figure V-1. If all of the processes are DELAYS and there is only one message in the loop, then the wait time at each input port is the time required for the message to traverse the loop when there is initially only one message. The problem is more difficult when there are initially several messages in the loop and when the processes are more complex. Simulation is used to answer such questions.

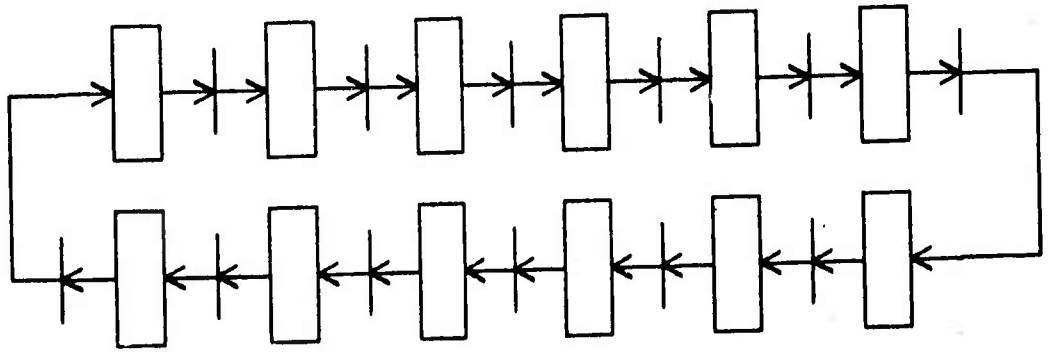


Figure V-1. A ring of processes

Queueing theory can be used to solve the same question and, in fact, to produce a more exact result, given assumptions based on known or estimated probability distributions [Kleinrock 75]. However, there are seemingly simple program

structures modeled by STEPPS that are difficult to solve using queueing theory. Each queueing model must be solved individually and the applicable techniques do not always transfer when systems with as many parameters as a STEPPS model has are involved; these limitations of queueing theory are well known [Fishman 73, McMillan 68]. STEPPS has been designed to be applicable to a variety of system structures and to make analysis easy for a system designer. In particular, simulation allows rapid interactive experimentation with a number of alternative problem decompositions.

V.B. Simulation operation and data collected

In order to discover the answers to the questions discussed in the last section, it is necessary to collect a sufficient amount of data by simulating a modeled program. The approach taken in this research is to make it possible to collect as much data describing the operation of processes and links as might be expected to be useful. The implementation of the data collection facilities has been carefully designed to facilitate the incorporation of additional facilities so that other than built-in analysis can be used.

The operation of a STEPPS model was described in Chapter II. At every possible change of simulated state† of a STEPPS model it is possible to collect data. Thus the specific operation of a process, a link, and the process scheduler will be described below and the data collected at each operation change will be noted.

---

†State in this instance refers to the condition of the entire model, namely all processes, all links and the process scheduler.

V.B Simulation operation and data collected

V.B.1. Process activity and the data collected

A simulated process goes through a sequence of actions that represent its activity. At each action the simulation system records the most recent action and collects appropriate data concerning the action. The specific data collection points and data collected at these points are:

1. *Activating.* When a process is initialized all data collection facilities are also initialized.

2. *Accesslink.* A process enters a new state when it accesses a link. The data collected are (a) a count of the number of entries to this state, and (b) the time that the link access begins. The process then tries to access the link by performing a synchronization check to exclusively access the link (a "P" operation on a semaphore).

3. *Mutex.* The time that is spent waiting on the link's exclusive access semaphore is accumulated. Next the process must guarantee that the link will be able to recognize the access. This is accomplished by means of a mutual exclusion between the link and the process.

4. *Startio.* The time waiting for the link to allow access is accumulated and the process performs its link access. The process then waits for the link to respond. The time for initializing this wait is recorded.

5. *Ioready.* The time spent waiting before the I/O operation can occur is accumulated.

6. *Iocomplete.* The time spent performing the I/O operation is accumulated.

7. *Endio.* The time when the I/O operation is completed is recorded. When a process state is repeated, steps 2 through 7 are repeated until all activity relating to the current state is complete.

8. *Choose.* The process then chooses which will be the next state. No data are collected at this point since this operation takes no time. When the operation of the model is traced this change of state is noted.†

9. *Computing.* The start of a process compute time is recorded.

10. *Endcompute.* The time spent computing in the current state is accumulated.

11. *Restarting.* The process is ready to be restarted and must be scheduled. The time is recorded.

---

†See Appendix A for description of using the simulator and tracing a simulation.

12. ***Readied.*** When the scheduler allows the process to proceed the time spent waiting while ready to run is collected.

Activities 2 through 10 represent a more detailed description of the operation of a process than is described in Chapter II. Steps 11 and 12 accumulate data concerning the time a process spends waiting to be scheduled. Analysis of the data is discussed below.

V.B.2. Link activity and data collected

The sequence of actions that a link goes through represents changes in the link's queue size, number of message requests, and time used by the link (if any). At each change, the simulation notes the new activity and collects appropriate data concerning the change. The activities of a link are:

1. ***Inactive.*** A link will be inactive until it is accessed. The time when the link becomes inactive is recorded.

2. ***Inaccess.*** A link has been accessed by an input port. Accumulate the amount of time between input request accesses and count the number of accesses.

3. ***Outaccess.*** A link has been accessed by an output port. Accumulate the amount of time between message available accesses and count the number of accesses.

4. ***Exclude.*** The link has been accessed and the time it was inactive is accumulated. The link now prevents any other access to itself by means of mutual exclusion synchronization.

5. ***Accessed.*** The time the link waited to exclude other accesses is accumulated.

6. ***Starting.*** If the link had to be restarted, the number of restarts is accumulated.

7. ***Started.*** The time after restarting is recorded.

8. ***Qlimit.*** If the link has no more room for messages (its queue size limit has already been reached), then the number of overflow messages is accumulated over time. This is the average number of processes that had to wait.

9. *Accept.* If the link can accept a message then the queue length is accumulated over time.

10. *Endaccept.* The time after accepting a message is recorded.

11. *Arequest.* The number of current requests is accumulated over time.

12. *Xmit.* If a message can be transmitted to a process requesting a message then the number of messages in the queue and the number of current requests are each accumulated over time.

13. *Endxmit.* After a message has been transmitted the time is recorded.

14. *Rereceive.* If a process had been waiting to send a message to the link, but could not, due to the link's queue size limit, then it is allowed to continue. The number of processes waiting to send a message is accumulated over time.

The activities listed cover all of the activity of a link. Data are collected concerning each property of the link that changes.

V.B.3. The scheduler and sets of concurrent processes

The function of this set of data collection facilities is to provide information that can be used to infer how the processes interact with each other over time. One measure is the average number of active processes. Another measure is concerned with which processes are active at the same time as other processes. Before a process becomes active it is scheduled to run by a process scheduler. Likewise, whenever a process becomes inactive, i.e. is waiting for some reason, the scheduler is notified.

The simulator is used to estimate the effects of restricting the number of processors. This restriction brings about the problem of the scheduling of processes when more processes are ready to run than there are processors able to run them. The STEPPS system provides the following scheduling algorithms:

1. First-in-first-out priority (FIFO). This algorithm schedules the process that has been ready for the longest time. When several processes have

been ready for the same length of time, an arbitrary choice is used to determine which one will be scheduled first.

2. PROCESS priority. Each process can be assigned a non-negative priority. When a choice must be made among ready processes, then the process with the highest priority number is scheduled first. When several processes have the same priority, FIFO is used to resolve the choice.

3. LINK priority. Those processes that are ready to run and are in an input state are examined first. These processes are requesting a message from a link. The process that is requesting a message from the link containing the most messages will be scheduled first. FIFO is used to resolve any additional choices.

4. PRLK priority. This is a combination of 2 and 3. After those processes with the highest priority are selected, then the process requesting a message from the link containing the largest number of messages is scheduled. FIFO is used to resolve any additional choices.

5. LKPR priority. This is another combination of 3 and 2. First the processes requesting messages from the links containing the greatest number of messages are chosen. Then the process with the highest priority is chosen among them. FIFO is used to resolve any additional choices.

6. RANDOM. A random choice is made among the ready processes.

These algorithms were chosen for inclusion in the STEPPS simulator because they are simple and have counterparts in real systems. The last-in-first-out algorithm was rejected because it does not adequately represent continued processing. Modifications to the STEPPS system that could include different scheduling algorithms are discussed in a later section of this chapter.

The data collected by the scheduler are listed below:

1. *Start.* A process is ready to run. Accumulate, over time, the number of active processes and the number of ready processors.

2. *Runaprocess.* A process is activated. Accumulate, over time, the number of active processes and the number of ready processes.

3. *Allactive.* A process is ready to run, but all of the processors are active. Accumulate, over time, the number of active processes and the number of processes ready to run.

4. *Startprocess.* A process is about to become active. For each process that is running collect the time. This represents processes starting to execute concurrently.

5. *Stopprocess.* A process becomes inactive. For each process that is still running accumulate the time that the two processes were running together.

6. *Deschedule.* A process has become inactive. Accumulate, over time, the number of active processes and the number of ready processes.

The data concerning the number of active processes are always collected, but the data concerning which processes are active concurrently are only collected when optionally requested.

V.B.4. Analysis of the data

For each process the total time for each activity and wait is accumulated. Performance expectations are computed for each of the following:

Percentage of time spent computing in each process state.

Percentage of time spent waiting to exclusively access a link for each state.

Percentage of time spent waiting until the link was ready to acknowledge access for each port.

Percentage of time each state waited until the link could complete the required I/O operation.

Percentage of time spent performing the I/O operation for each state.

Percentage of time the process was ready to run but had to wait to be scheduled.

For each link the following performance expectations are computed:

The percentage of time the link was active, inactive, and restarting.

The percentage of accesses required for the link to restart.

The expected time between link accesses, between input port accesses, and between output port accesses.

The expected queue length.

The expected number of processes waiting to send a message to the link.

The expected number of processes waiting to receive messages from the link.

The analysis of the schedule data is used to compute:

The expected number of active processors.

The expected number of processes that must wait to be scheduled to run.

The fraction of time each process computes concurrently with each other process.

Answers to the questions presented in the first section of this chapter are all available from this analysis. Estimates are available concerning all of the activities of a process and a link. Bottlenecks in the system occur at those links where queue lengths are large and where processes are forced to wait for reasons other than the completion of an input or output operation. By examining the number of active processors, decisions can be made concerning numbers of processors needed for the program. Data concerning the working set of processes can be used to facilitate prescheduling of sets of processes. Likewise when processes are known not to run concurrently it is possible to manage data resources to take advantage of this occurrence.

For the simulations presented in Chapter III, a variety of the STEPPS simulator variables, data collection, and data analysis facilities proved useful. The Bliss/11 experiment emphasized varying the number of processors available and using the alternate scheduler algorithms: FIFO, LINK, and RANDOM. The specific data collection and analysis facilities that were the most useful included:

number of messages into and out of each link,

expected queue lengths at each link,

expected process wait time at each link and process ports, and

average number of active processors.

The Hearsay II experiment was more complex, and used additional STEPPS simulator facilities. The "working set of processes" analysis was used to determine the proper number of Knowledge Sources to reproduce. Since the overhead associated with this facility was large, it was not used beyond the system tuning simulations. The other facilities that were utilized during the simulations were:

link queue lengths used to show where interference,

percent of time spent in process states used to observe which processes contributed to the link queue lengths, and

average number of active processors.

V.C. The implementation of the STEPPS system

The STEPPS interactive system has been implemented on the Digital Equipment Corporation PDP-10 computer. It is constructed using the Sail [VanLehn 73] and Bliss/10 [Wulf 71] programming languages. These languages were chosen since each contains features that are most appropriate for its use. The discrete time simulator uses a modification of a package of Bliss/10 (hereafter referred to as Bliss) programs called POOMAS (Poor Man's Simula) originally written by A. Lunde [Lunde 71]. The STEPPS system consists of about 45,000 words of PDP-10 36-bit word memory.†

The STEPPS system command language was designed with user convenience in mind. The command syntax consists of three types: node connection, transition matrix manipulation and keyword commands. Wherever possible, unique abbreviations are acceptable. For example, ALPHA.1←BETA.1 means to connect port ALPHA.1001 to a uniquely named new link (say Link003) and then connect this new link to port BETA.0001. Another example, DIS CON ALPHA, LINK003, BETA.01 is the same as

---

†This includes about 10,000 words for a debugging package and library.

DISPLAY CONNECTIONS ALPHA, LINK003, BETA.0001 which displays the connections to the objects requested. Every parameter to a STEPPS model can be displayed and modified by one or more commands. An annotated protocol of examples using the STEPPS system is presented in Appendix B.

The interactive portion of STEPPS was written in Sail and takes advantage of Sail's powerful string manipulation and input/output facilities. The lexical and syntactic analyzers for the STEPPS commands are written in Sail. The data structures representing a model are maintained by a set of Bliss programs. The Sail program that performs the interpretation of the STEPPS commands is recursive, so that when a command to LOAD from a PDP-10 file is given, the system simply calls the main interpret program recursively. This means that commands in files can cause other files to be loaded. The displays of the STEPPS model components are in the same form as the command language. Thus the display of the STEPPS components can be sent to a file and later read in as a set of commands. The Sail portion of the system acts as a front end to the Bliss portion of the system.

The Bliss portion of STEPPS contains programs which create and manipulate a representation of a STEPPS model. The representation consists of a complex data structure where each link node and process node contains pointers to other nodes, as in the directed graph representation of a program model. The use of pointers and complex data structures is one of two reasons for choosing Bliss to implement the representation of a STEPPS model. The other reason is the availability of the POOMAS simulation package for Bliss programs.

The internal data structures are complex since the STEPPS system allows a wide variety of manipulations of a model. A process is created when it is first used, either to define a connection between a port of the process and a link or to create a

transition matrix for a port of the process. Subsequently, additional ports can be added to the process, new connections made, and changes made to the transition matrix. Whenever a modification that affects the transition matrix occurs, a validity test is performed to insure that the matrix contains proper probabilities (i.e., with rows summing to one). This prevents improper alteration of the transition matrix and sometimes prevents the removal of a port of a process. A link is created in the same manner as a process, viz. when it is first used for a connection or when it is assigned attributes.

The model simulator is constructed from three types of POOMAS simulator processes: STEPPS processes, STEPPS links, and the STEPPS schedules. The operation of these simulated processes has been described earlier (section V.B). There are pointers in the STEPPS data structures that go from the simulation representations to the STEPPS representation and vice-versa. This facility makes it possible to examine the progress of a simulation and later continue the simulation. The data collection facilities are localized and this enables ease in adding to or modifying any of these facilities. The data analyzer functions are also localized which also makes it easy to add to or include other analysis facilities.

The speed of the simulator is measured by the number of events per second. The events are: link access, link startup, link delay, process perform I/O, process start computing, and process stop computing. Other states of a process and a link do not cause the simulator to schedule an event. The time consumed by the scheduler is not measured in terms of events, but is included as the overhead for process scheduling. The resulting measured speed is approximately eighty events per second. An estimation of the length of time required to obtain results concerning a model depends on the complexity of the model. The STEPPS system maintains counts of the

occurrence of various events and so it is possible to examine whether enough events have occurred to continue or discontinue a simulation.

The remaining major component of the STEPPS system is the deadlock recognition algorithm, which is also written in Bliss. The general algorithm has been described in Chapter IV. The technique used is to iterate through the set of links and apply reductions R1, R2 and R3 to each process attached to the link. The links and processes attached to them are examined repeatedly until none of the reductions is applicable. Finally reduction R4 is applied to remove the remaining SOURCE/SINKS. Actually, whenever a SOURCE/SINK is identified and all adjacent ports to it have been removed, the SOURCE/SINK is removed as well so that it need not be examined on each cycle through the graph.† In addition, once the last connection to a link is removed, the link is also removed. Although the order of application of the reductions is unimportant, as far as the ultimate result is concerned, the following is the order chosen for implementation, and reasons for choosing this order:

1. R2c (remove ports attached to SOURCE/SINKS). This reduction is expected to cause the largest number of reductions to occur. It is also an easy condition to determine.

2. R2a (remove ports of the same type and processes from the same link). The conditions for this reduction are easy to determine and reduce the number of ports attached to the link.

3. R2b (remove ports of different type and same process from same link).

4. R1 (combine adjacent processes).

5. R3 (combine parallel processes attached to the link). This reduction is the one most likely to benefit from application of the other reductions.

Since each reduction removes one or more connections, the total number of reductions is at most the same as the number of ports, which equals the number of

---

†The algorithm description (Chapter IV) was simplified by not including this implementation alternative.

connections. A more interesting measure of the cost of the reduction algorithm is the number of ports that must be examined. The worst case would be one successful removal of one port per examination of the ports. If there are N ports in a graph, then the algorithm would require at worst N! port examinations. A more realistic estimate should be based on the successful application of more than one reduction per pass over of the ports. If one fourth of the ports are removed per loop through the ports† then the total number of examinations required is approximately 4*N. The reason that this estimate is more realistic is that successful application of a reduction at the beginning of a loop through the ports can cause the application of a reduction that might not have occurred before.

The STEPPS system was designed so that it would be possible to include analysis programs that are not original components of the STEPPS system. An example might be to use an analysis of semi-Markov processes. The STEPPS system will allow such a program to be written in FORTRAN, Sail or Bliss and later included with the STEPPS system. The method is to link the new program with the STEPPS system and then apply the new program to a STEPPS process. The STEPPS system will convert the internal representation of a STEPPS transition matrix to the form expected by a FORTRAN or Sail program (i.e., a matrix) and then perform a call of the FORTRAN, Sail or Bliss program. The structure of the transition matrix is defined to be the same as displayed by the STEPPS "DISPLAY" command. Another type of analysis that might be written externally to the STEPPS system is the analysis of a connection matrix representing the entire STEPPS model. For this situation, a matrix will be formed to represent the connections among the processes and the application of the external analysis program would be performed on the representative connection matrix.

---

†All tested cases resulted in even greater reductions than this.

Even though the STEPPS system was designed to accommodate externally defined analyses, the easiest way to include new features into the STEPPS system would be to add them to the system itself. This task should not be difficult for a Bliss programmer, since the system is well organized into many small subroutines, and is internally well-documented. Very few of the routines in either the Sail portion of STEPPS or the Bliss portion are more than fifteen lines long, so their complexity is kept to a minimum. In addition, the system includes a large number of debugging facilities. The removal of the debugging facilities would probably decrease the size of the STEPPS system by about twelve thousand words (this includes eight thousand words of non-STEPPS debugging tools).

The STEPPS system, as constructed, is really a prototype for tools that should be available to a systems designer. As such, a number of lessons were learned concerning the systems implementation. One criterion adhered to was the emphasis on man-machine interaction convenience. Many times the ease of using simple, yet descriptive commands made the STEPPS system appear elegant even when features were being debugged. In a successor system, even greater emphasis should be placed on the man-machine interaction than in the prototype system. The amount of extra code and nominal extra processing time are well worth the user convenience. The STEPPS structure was noted above as being well organized, which also must be emphasized as a valuable lesson. It was often found that disciplined programming style used and appropriate testing and debugging aids constructed greatly assisted in the overall system development.

There were implementation drawbacks in addition to the constructive lessons. The STEPPS system uses a set of fairly complex data structures. It was not estimated during the system design that these structures could grow rapidly (eg. whenever a

new port was added). Thus during the application of the STEPPS system to the examples of Chapter III, the data structure had to be redesigned and rebuilt. Fortunately the previously mentioned programming discipline used made this somewhat painless in terms of propagating errors (some "information hiding" had been used). It must still be observed that the data structure problem is not solved, but could be if the 'next version of STEPPS handled simulations and model structures in a fashion different from the current system.

A similar improvement can be made to the STEPPS system by constructing a discrete event simulator tailored to the STEPPS model. The POOMAS simulator was used for convenience, but it contains unneeded features that add to the STEPPS system size and add to the time required for a STEPPS simulation. Thus the simulator should be optimized for STEPPS simulations.

Chapter VI

Summary

In this thesis, the problem of designing programs for asynchronous multiprocessor computers has been addressed. A particular design philosophy has been emphasized consisting of predicting the implications of design decisions at early stages during multiprocess program design and development. The thesis presents design tools consisting of a model for describing the decomposition of a program into asynchronous, concurrently executable subparts; an analysis algorithm to determine whether a model contains a deadlock; an interactive system for manipulating a model representation; and a simulation system for predicting the performance of program structure under a variety of scheduling algorithms.

These tools (called STEPPS) have been used to model possible program structures with fine granularity (as with Petri nets) and at a functional level. Potential structural problems may be identified and a program restructured before an investment is made in a poorly structured program. Two experiments have been performed to predict performance implications of multiprocess structures. In one experiment, using a STEPPS model and the STEPPS system, the implications of restricting the numbers of available processors and using different scheduling algorithms were examined, and the effect of using alternate program structures was explored. In the other experiment it was shown that, when a multiprocess program under development is sufficiently instrumented, the STEPPS model and system can be used to help tune the program's structure.

Thus it has been demonstrated that the STEPPS model and the STEPPS system do help to accomplish a well structured design.

VI.A. Designing Programs for Multiprocessor Computers

The past few years have seen the advent of multiprocessor computers (See Chapter I), and more are being developed as hardware costs decrease through technological advancement. In addition, since microprocessors and mini-computers are being connected to comprise new multiprocessor networks, the need has arisen to design programs to utilize these multiprocessor computers. It is now recognized that the total cost of a computer system has become based more on software costs than on hardware costs [Boehm 73]. Through proper software design the costs for testing, coding, debugging, redesigning, maintaining, and extending software can be better controlled, thereby decreasing the total cost of the computer system. The software design tools discussed within this thesis are particularly valuable due to the current interest in multiprocessor programs.

The approach taken for understanding how multiprocess program components interact is based on the interprocess communication structure. It is at this level that an abstraction can often be made for a system. Central to the abstraction is the decomposition of the total system into a suitable set of functional components. Consequently, understanding how a multiprocessing system works can be aided by understanding how the components of the system communicate.

Several tools have already been developed as aids to the design and analysis of multiprocessor systems. Of these tools, modeling techniques used include Petri-nets [Petri 62], the UCLA model [Estrin 63], and queueing theory models [Kleinrock 75]. They have been used to represent interprocess control and data flow, program validity, bottleneck identification, and program determinism. However, these models suffer from being so complex or abstract as to not really represent the functional

aspects of the total system. In addition, these models often have been difficult to analyze. Another approach was taken by Riddle [Riddle 72], who combined a functional structure with program-like descriptions of individual processes. This model was an improvement in understanding overall system structure, but it still suffered from requiring programming detail to describe a process's interactions. Similarly due to its program-like ature and to its complex algebraic form, Riddle's models are difficult to analyze. Another type of tool, simulation, has proven to be a valuable approach to analyzing system design. However, simulations must be individually programmed in a suitable programming language (e.g. GPSS or SIMULA). Simulation models provide much useful information, but like most programs they are difficult to construct and (often) to modify.

STEPPS consists of a set of design tools that combine several of the advantages of the abovementioned tools with a new idea felt to be natural for system design. The major new concept is that processes comprising a multiprocess program are abstracted as operating in a probabilistic manner with respect to their interprocess communication activities. The STEPPS system was designed to avoid the dual problems of very fine required detail and reprogramming for examining implications of alternate multiprocess program structuring. The other features of the STEPPS tools comprise an interactive system used to simulate, manipulate, and analyze STEPPS program models.

VI.A.1. The STEPPS system

A STEPPS program model is a directed graph consisting of two types of nodes: process nodes and link nodes. Communication among nodes in the model is represented by the movement of message tokens. The operation of the entire model is

defined in terms of the individual operation process nodes. A process can request messages from and send messages to link nodes. The sequence of operations of each process is defined similarly to the operation of a semi-Markov process. That is, both a probability and a computation time are associated with each possible successive process operation (request a message from a link or send a message to a link).

The STEPPS model is more expressive in terms of modularity and potential activity than Petri-net like models. Yet the STEPPS model abstracts many of the expressive details provided in programming languages and programming-like models. The model is at an abstraction level that emphasizes both interprocess communications and internal process complexity based on probabilities and timing. In Chapter I it was demonstrated that the STEPPS model could incorporate both the Petri-net model and the UCLA model. In Chapter III, more natural examples were also demonstrated using STEPPS: fork/join, subroutines, probabilistic server processes, P/V, and reader/writer. More importantly, two non-toy, more complete examples were modeled and simulated: Bliss/11 and Hearsay II.

The STEPPS simulator, which is invoked from the STEPPS system, can be configured to represent a variety of execution environments. These environments are defined in terms of the number of processors available and scheduling algorithms used when a scheduling choice is required. Data are collected and analyzed to predict such aspects of a modeled program's performance as queue lengths, rates of data flow at links, process activity and parallelism. Some measures of parallelism of interest to STEPPS are those concerning the average number of active processors and the working sets of processes.

The STEPPS interactive system was designed to facilitate man-machine interaction. Some features of the system are: commands for creating and manipulating

STEPPS model representations; commands for saving a model or parts of a model for later retrieval; commands for displaying all parameters of a model and simulation; abbreviations for most commands; and automatic assignment of default values to unspecified model and simulation parameters.

The STEPPS model can be automatically analyzed to determine the existence of a deadlock possibility. This analysis (detailed in Chapter IV) is performed by iteratively applying a sequence of graph reductions to a STEPPS program model until no further reduction is applicable. The reductions, which are applied when certain constraints are satisfied, are: combine two adjacent processes into one; eliminate states or combine two states to be one state of a process; combine two processes that are in-parallel; and eliminate processes that can perform only one operation.

It has been demonstrated that each reduction preserves the possible interprocess communication among those processes not involved in the reduction. If as the result of the completion of all possible applications of the reductions there are no nodes remaining then the original representation was that of a structure with no deadlocks. Otherwise the original structure contained a non-zero probability of a deadlock. The complete reduction algorithm has been implemented as part of the STEPPS system.

VI.B. Experiments and Results

Two non-trivial experimental applications were conducted to validate the usefulness and significance of the STEPPS tools. The Bliss/11 compiler structure [Wulf 75a] was studied, modeled, and analyzed for reconstruction on a multiprocessor based upon a design similar to its sequential structure. The Hearsay II multiprocess speech

understanding system [Lesser 74, Fennell 75b] was analyzed using STEPPS to help explain a phenomenon of interprocess interference [Fennell 75a, 75b].

VI.B.1. The STEPPS Bliss/11 application

The STEPPS Bliss/11 application, presented in Chapter III, was performed to predict potential throughput increases if the compiler structure were moved to a multiprocess organizational environment. Using a multiprocess structure, based upon the compiler's pipeline organization [Wulf 75a], it was found that throughput could increase approximately 3.5 times over throughput achieved with a single process structure. In addition, by using the STEPPS simulator features to restrict the number of available processors and schedule ready processes on them, it was found that most of the throughput increase could be attained by using two thirds of the potential number of processors. Furthermore, it was found that varying scheduling algorithms available to the STEPPS system (i.e. FIFO, most waiting requests, and random) did not appreciably affect the throughput rate.

The Bliss/11 structure was augmented to examine the consequences of providing duplicate processes for some of the Bliss compiler components. The results of the simulations demonstrated that there would be an increase in throughput, but that potentially it was not large (about 4 times sequential). These results also indicated that the necessarily sequential lexical analysis stage of the compiler is a significant bottleneck preventing compilation speedups. Again it was observed that most possible throughput was reached by using about two thirds of the potential number of processors.

A systems designer embarking on designing a multiprocessor Bliss/11 implementation can use these results to aid in determining where to concentrate

efforts. It was predicted that the crucial part of the compiler process structure was the purely sequential lexical/syntactic analysis component. Hence this component should receive attention to optimize its processing. Alternatively, it could be concluded that there is processing time available to perform more sophisticated and time consuming semantic optimizations since the lexical/syntactic analysis uses the bulk of the compiling process. Another conclusion for the systems designer is that there does not appear to be a large gain in processing achieved by designing a compiler to dedicate a processor to each process. Instead it appears that a design based upon fewer processors than the potential number of processes can achieve almost as good a throughput rate.

VI.B.2. The STEPPS Hearsay II application

The STEPPS Hearsay II application, presented in Chapter III, demonstrated that STEPPS can be used to model abstractly a real multiprocess program structure; to reproduce an interesting phenomenon of that program structure; and consequently to indicate whether the cause of the phenomenon is at the structural level abstracted by the STEPPS model. The Hearsay II Speech Understanding System (HSII) [Lesser 74, Fennell 75b] has been designed to utilize a variety of analysis sources to solve the problem of understanding human speech for the performance of a task. The task has been functionally decomposed in a data driven structure so that individual components of the understanding process can be performed concurrently in a closely-coupled multiprocessor environment.

The STEPPS model was used to represent the operation of the individual processing components of the HSII system: the ***precondition*** (PC) processes and the ***knowledge source*** (KS) processes. In addition, the data base (DB) ***blackboard*** was

modeled as a set of synchronization locks. This model is an abstraction based upon an analysis of the HSII structure and upon data provided as a result of instrumentation incorporated into the HSII system by the HSII designers and implementers. The specific data provided were obtained from executing HSII in a sequential (single processor) manner.

The STEPPS HSII model is probabilistic in nature and is based on potential communication activities. The three types of communication activities emphasized are: initiate a precondition process, access the data base, and initiate a knowledge source process. The data provided from the implemented prototype HSII system were analyzed to provide estimates for choices of precondition processing activity. The data were also used to determine STEPPS HSII process computation times and probabilities for accessing portions of the data base. Probabilities (based on the provided data) also are used to indicate a precondition's potential initiation of a knowledge source process.

The accessing of the HSII data base blackboard is organized as a hierarchical (lock/unlock) synchronization structure to maintain data integrity and to prevent processing deadlocks. Fennell [Fennell 75a, 75b] performed simulations of a multiprocessor HSII system and discovered that locking interference placed a substantial overhead on the HSII throughput rate as measured by the average number of active processors. Specifically, he found that the interference decreased processing by about two thirds, but he did not explain the reason for this phenomenon.

The STEPPS system was proposed as a tool to determine whether the locking interference phenomenon occurred due to locking of a small number of data base segments or whether the problem was more complex. An implication might be that the locking hierarchy mechanism might be made simpler, i.e. less finely grained.

Simulations were performed, based on the STEPPS HSII model, varying the number of possible data base regions that could be locked. In addition, the model was simulated with locking turned off (as in Fennell's simulations). The result was that the average active number of processors with and without locking, using the STEPPS simulations, corresponded to Fennell's results.

This result is significant in that the locking phenomenon was reproduced while based on a simple probabilistic communications model. Since the probabilities used were taken from a sequential execution of the HSII system, the interprocess cooperation did not seem to affect greatly the locking interference problem. The hypothesis that the interference was due to locking of a small number of regions was supported by the simulation statistical results.

The STEPPS system was demonstrated as providing the HSII system designers with a tool for modeling communications structures. It has provided the HSII designers some interesting information, and through modification of the computation times, probabilities, and model structure it should be able to provide more information. Thus the STEPPS HSII model should be a useful framework for exploring the effects of possible design changes suitable to the model's structure.

VI.C. Future research and refinements to STEPPS

The STEPPS model and simulator are based on a fixed interprocess communication pattern and a fixed number of processes. These restrictions were judged to be necessary when the deadlock detection algorithm was designed. It is unknown whether dynamic creation and deletion of processes will still allow the application of a deadlock detection algorithm. It may be necessary to restrict the types of operations and connections that dynamically created processes can have.

Several other generalizations of interprocess communication may be considered as modifications to the model. Hierarchical and interrupt relations are multiprocess relations that it is not now possible to model using STEPPS; however, it may be possible to modify the model to include such structures.

A limitation of the STEPPS model discovered during applications of the model was the introduction of extra modeling complexity required to model some possibly interesting program communication structures. For example, the STEPPS reader/writer model demonstrated that the STEPPS model only worked with a finite number of readers/writers. This is a symptom of processes' actions not being determined by information carried by the message tokens (e.g. tagging, sender, return-request, etc.). The inclusion of actions (other than timing) based on message contents would add program-like complexity to a process model and would also discount the present formulation of the deadlock detection algorithm. Any extension to the STEPPS model based on including message information may not prove fruitful due to its own form of added complexity.

Areas of deadlock analysis beyond the detection algorithm would be the identification of cause(s) and the prediction of the probability of a deadlock over time, events, or some other measure. The STEPPS system can be used to trace the application of the deadlock detection algorithm and to display a resulting irreducible graph. Studying this trace and the resulting graph has proven useful in discovering the cause of deadlocks while testing the STEPPS system, and it may be possible to create an algorithm for this process.

The STEPPS interactive system uses simple linear displays of a STEPPS model. It may be possible to create more natural displays of the directed graph representation of a model. This problem may be difficult because a model has no defined root, terminal or topology.

As the STEPPS system was applied to the examples presented in Appendix B, features were added to enhance the convenient use of the system. It is possible that future experience using STEPPS will indicate other improvements. Some extensions that might be useful are:

1. Compute time between port activations need not be fixed; instead it can come from a definable density function. This is closer to the actual definition of compute times for a semi-Markov process.

2. The ability to identify a group of nodes and copy them in one operation may be used to organize similar subparts of a large structure. It is already possible to copy single nodes and transitions from a process state.

3. The state of a simulation could be saved for future continuation.

The STEPPS system has been shown to be useful in the design and analysis of two multiprocess programs, viz. Bliss/11 and Hearsay II. Now that the author has tested and debugged these examples on the STEPPS system, others should use STEPPS or a system very much like it in the complete design and construction of multiprocess programs. STEPPS is intended as a useful group of design tools and should be used for that purpose.

As experience is gained in using the STEPPS model, more techniques such as those presented in the beginning of Chapter III can be created. For example, other synchronization techniques may be designed, in addition to the PV and reader/writer examples shown.

Some problems which might be constructed for a multiprocessor and which could use STEPPS are: a multiprocess compiler implementation, sort and search programs, theorem provers, data pipelines (with and without feedback), and data base management programs.

VI.D. Conclusions

As noted in Chapter I, several multiprocessor computers are available and/or being developed. In particular, C.mmp [Wulf 75b] has reached a stage of maturity where several multiprocess programs are being designed and developed for implementation on it. The tools presented in this thesis research should be useful to those designing programs for C.mmp or for any communicating multiprocess program environment.

By using an interactive system, a program designer can create a model of his program and discover a variety of implications of his design decisions. The STEPPS system is most appropriate for this type of exploration of a program structure space. STEPPS provides analysis tools and simulation tools in one interactive system. Neither unique model analysis nor unique simulation models need to be developed when the STEPPS system is used.

A second advantage of using the STEPPS system concerns the ability to predict performance changes in a running system before making modifications to the system. This type of design decision is important for determining where to direct efforts to improve a system's performance. The overall structure of a program is no longer the only issue; instead considerations include the sensitivity of a program's performance to modifications of the program structure and changes in modeled probability and time parameters.

The major advantage of the STEPPS system over other systems analysis tools and techniques is that the STEPPS system automates the production of results. Furthermore, if methods of analysis that are not already available within STEPPS are of interest, it may be a simple task to include these other methods with the STEPPS

system. Finally, the Bliss/11 example demonstrated that STEPPS can be used to provide performance predictions quickly for a multiprocess program design, and the Hearsay II example demonstrated that decisions may be made concerning modification to an ongoing system, based upon a simulation of a STEPPS model of that system. Thus we conclude that the STEPPS interactive system is a useful tool for the design and analysis of multiprocess program structure.

APPENDIX A

STEPPS System Manual

This appendix contains a complete description of the STEPPS system facilities and their use. For clarity, more than precision, BNF notation is used to describe the command syntax.

A.1. Introduction

The STEPPS system is an interactive system for use in modeling and simulating multiprocess programs. The following services and facilities are provided:

Creation and manipulation of models

Displays of all model constituents

Analysis for well-formed and deadlock-free model

Simulation and data collection

Display of simulation parameters, state, collected data, and statistics

Model description saving and retrieving

The three distinctive types of commands are: set model connections, define transition matrices, and keyword. These distinctions exist for user syntactic convenience.

The model connection command is recognized by its inclusion of at least one "←" and is used to connect model nodes. The transition matrix command is recognized by its inclusion of one "="[†]. The keyword commands begin with a command keyword and never contain "←" or "=".

[†]The one exception to this will be explained.

Names of objects, processes or links are defined as in most languages with the restriction that a maximum of 10 characters can be uniquely distinguished.

<name> ::= <letter> | <name><letter> | <name><digit>
<letter> ::= A|B|C|D|E|F|G|H|I|J|K|L|M|N|O|P|Q|R|S|T|U|V|W|X|Y|Z|*|$|"|"
<digit> ::= 0|1|2|3|4|5|6|7|8|9
<process name> ::= <name>
<link name>::= <name>

While all input to the STEPPS system may be either upper or lower case letters, lower case is automatically converted to upper case. Thus lower case names can be used for convenience, but they are indistinguishable from upper case names with the same characters (and order).

Process ports are identified by the process name, the port type, and the port number. The usual definition is:

<port name>::= <process name>.<port type><port number>
<port type>::= I | O
<port number> ::= positive integer less than 1000
<untyped port name> ::= <process name>.<port number>
<port id> ::= <port type><port number>

Some connection commands allow abbreviations for <port names> using <untyped port name> and context for definition.

Keyword commands begin with a keyword and parameters follow on the same line. The actual syntax of the keyword parameters is dependent on the particular keyword. However, consistency among some keyword parameters is that keyword subparameters are usually order independent. Also, keyword abbreviations and parameter-subkeyword-abbreviations can be used by entering unique initial character strings. Thus E may be used for EXIT, but DI must be used for DISPLAY since DENSITY is also a command.

Spaces (at least one) are used as separators between keywords and parameters. In some situations a comma may be used instead of a space, but a space

can not be used in place of a comma. Spaces may be freely inserted around separator characters ":;,←=[]". Comments can appear on any line by placing an "!" which causes everything to its right to be ignored. Line continuation is used by placing a "-" as the last non-comment character on a line. Thus,

```
DIS-! first line <cr>
PLAY-<cr>
 GRAPH ! 3rd line<cr>
```

is a legal command†.

Each parameter that can be set by commands has a defined default value. These default values will be presented in the command descriptions.

For the BNF syntax descriptions, two notational assumptions will be used. A syntax root, a list definition, and a command will all be assumed. Thus the following describes the missing syntax:

```
<STEPPS commands>     ::= <connect nodes> | <define matrix>
            | <keyword command>
<keyword command>     ::= <"1st" keyword command> | <"2nd" keyword command>
            | <"3rd" keyword command> . . .
<y-list>      ::= <y> | <y-list>, <y>
<x params>  ::= <x param> | <x params> <x param>
```

A.2. Model Creation

A model is created by defining the connections among its nodes, its transition matrix values, and its link attributes. A model can be given a name by using the command MODEL. This name is used when displaying the model components, and when saving and retrieving the model description.

---

†<cr> means carriage return.

A.2.i. Connecting nodes

The following is the syntax for connecting nodes:

<connect nodes> ::= <port connection> | <link connection>
<port connection>:: <typeless port list> ← <connect nodes>
<link connection> ::= <link name> ← <port connection>
<typeless port> ::= <process name>.<port number>
<port number> ::= non-negative integer less than 1000

A connection between an input port and a link is represented by: the input port name, then a left arrow, and then the link name. In place of a single input port, a list of ports can be used to denote that each port in the list is connected to the link. Contrary to the above BNF definition, the type of port can be included (i.e. input port). Also, when several ports of the same process are to be connected, the process name may be left out after appearing once. The following are legal connections:

a.1 ← alpha
b.i1, c.i2 ← beta
d.1, a.2,.3,.i4, b.i7 ← alpha

The results of these lines would be to connect input port A.I1 to link ALPHA, input ports B.I1 and C.I2 to link BETA and to also connect input ports D.I1, A.I2, A.I3, A.I4, and B.I7 to link ALPHA. A.I1 will remain connected to link ALPHA.

A connection between a link and an output port is represented by: the name of the link, then a left arrow, and then the name of the output ports. In place of a single output port, a list of ports can be used for connecting each to the named link. As above, the type of port may be included and process names need not be repeated. The following are legal connections:

gamma←d.2
delta←e.1,f.o3,.4,g.7

The two types of node connections can be combined. When a link appears between sets of ports the meaning is that the input ports (to the left of the first left

arrow) are connected to the link and also that the output ports (to the right of the second left arrow) are also connected to the link. The following is used to connect the ports X.11, Y.12, Z.11, S.03, R.02, T.03 to the link GORP:

x.1,y.2,z.11←gorp←r.o2,t.3,s.3

Another method for combining connections is used to denote the connection of links to ports having the same number but different types. For example,

eta←p.3,q.7←nu

means the same as

eta←p.3,q.7
p.3,q.7←nu

Note that the ports must be typeless when using this notation since it represents connections to both input and output ports.

A generalization of the above is also allowed:

a.19←epsilon←f.3, l.7←kappa←c.3←omega

An additional notational convenience is available to automatically generate a unique link name. It is accomplished by using port names on either side of a left arrow. Thus,

b.3,c.2←a.421

means to generate a new link name (e.g. LINK017) and connect it to the ports used (B.3, C.2←LINK017←A.421).

A.2.ii. Setting transition values

The following is the syntax for setting transition values:

<set probabilities>::= <port name><repeat factor> = <initial flag><prob vector>
<repeat factor> ::= [<repeat number>] | null
<repeat number> ::= a positive integer less than 262144
<initial flag>::= * | null
<prob vector> ::= <prob seq> | <prob seq>; <prob vector> | <prob vector>;
<prob seq>::= <port prob comp>/<prob seq>

<port prob comp>::= <I or O><port number> : <prob comp>
<prob comp> ::= <prob>, <comp> | <prob> | , <comp> | <prob>,

A <prob> is a real number. If it is in the range [0.0,1.0] then it is the assigned probability. If in the range [2.0,3.0] it is a defaulted amount and is ignored. If it is negative then the value becomes defaulted. If it is larger than 3.0, then this is an error. <comp> is any non-negative real number.

To set transition probabilities the source port is written to the left of an "=". If the port activity is to repeat before a transition is made, then the repeat factor is placed within square brackets, between the port name and the "=". To the right of the "=" appears the destination probability; identified by the destination port type and port number followed by a colon and then followed by the transition probability, a comma, and the associated compute time. If a "*" occurs to the right of the "=", then the named port is designated as the initial port. The following are examples:

a.i2=o3:.5,1.6
b.o3[6]= o4:1.0,.1
c.I1 = * I2: .6,0.0

These lines mean that p(A.I2,A.03)=.5 and the related compute time is 1.6. Port B.03 repeats six times before entering state 04 (each time computing for .1). C.I1 is an initial port and p(C.I1,C.I2)=0.6.

Several abbreviations can be used:

1. Either the probability or the compute time can be left out.

2. The following is a sequence of state changes:

a.i3=o1:1.0,.5
a.o1=i2:.5,1.0
a.i2=i4:1.0
a.i4=o2:.2

This can be abbreviated as:

a.i3=o1:1.0,.5/ i2:.5,1.0/ i4:1.0/ o2:.1

3. More than one sequence or single change can be shown on one line:

```
b.o16=i5:.3
b.o16=i2:.1/o3:.5
b.o16=o4:.5
b.o16=i3:.1
```

becomes

```
b.o16=i5:3; i2:.1/ o3:.5; o4:.5; i3:.1
```

A.2.III. Model manipulation commands

Several keyword commands are used to manipulate a model representation. Their functions include creating link attributes, copying nodes, removing nodes or ports, and creating special types of process structures. The following is a brief description of these commands.

ATTRIBUTES is used to assign the link attributes to links. The specific attributes are maximum queue length, initial queue volume, start-up time, and delay time.

COPY is used to copy nodes based on an existing node. It can also be used to copy ports.

CLEAR is used to remove all processes and/or links.

DENSITY is used to connect a process to a link as if the process sent or received messages with a rate based on a given probability density function. The density functions available are exponential and normal.

DISPLAY is used to display model attributes at the terminal.

REMOVE is used to remove individual processes, ports, and links.

A special link called DANGLING is the default connection to any unconnected port. Explicit connections can be made to DANGLING, but a model will not be well-formed if any connections remain to it. DANGLING can not be removed.

A.3. Model Analysis and System Commands

Keyword commands are used to analyze and test a model. In addit. ı, there are STEPPS system commands used to interact with the underlying PDP-10 operating system.

APPLY is used to apply a function that is defined external to the STEPPS system to either individual processes or to an entire connection matrix.

TEST is used to test for a well-formed model. It is also used to test whether a model is deadlock-free.

EXIT is used to exit from the STEPPS system and reenter the PDP-10 operating system.

LOAD is used to retrieve STEPPS commands from a PDP-10 file.

SAVE is used to save the representation of a model onto a PDP-10 file. The representation is in the form of commands to recreate the items saved.

A.4. Simulation commands

The simulation features of the STEPPS system allow for the assignment of several parameters. Most of the simulation parameters can be displayed and altered independently of the invocation of the simulation. The parameters are concerned with scheduling, data collection, and tracing. A model can be simulated for a period of time and then a snapshot can be taken of its current state. Statistics can be displayed and the simulation may be continued. No alterations can be made to the model while a simulation is in progress and the STEPPS system prevents this from happening by asking whether the modification should really be made. If so, the simulation is terminated.

COLLECT is used to mark the processes that will and will not have data collection.

CONTINUE is used to continue a stopped simulation. It also can be used to turn on simulation tracings.

DISPLAY is used to display the simulation parameters.

SCHEDULE is used to assign the simulation scheduling algorithm, to mark process priorities, and to mark which processes are and are not competing for processors.

SIMULATE is used to invoke the simulator. Some parameters can be assigned using this command. In addition, tracing can be turned on by the command.

SNAPSHOTS is used to display the status of process nodes, link nodes, and/or the scheduler when a simulation is stopped.

STATISTICS is used to display collected data with analysis for process nodes, link nodes, and/or the scheduler when a simulation is stopped.

UNSIMULATE is used to terminate a simulation that has stopped. Once this command is used, the simulation can not be continued.

A.5. Keyword commands

The following is a detailed description of each of the STEPPS keyword commands. The commands are given in alphabetical order. Parameters are described with each of the commands.

APPLY

<APPLY cmd> ::= APPLY <external function name><APPLY param>
<external function name>::= <six character name>
<APPLY param> ::= GRAPH | PROCESS <list of process names>

The named function is applied to either the entire GRAPH or to the transition matrix of each named process.

The method for incorporating an external function with STEPPS depends upon the language used for the function: BLISS, SAIL, or FORTRAN. SAIL and BLISS are the most appropriate languages to use since the use of FORTRAN requires some restrictions (I/O can only be performed by using SAIL procedures). The following are the required procedures to use a function GORP defined in different languages.

BLISS:

1. Define GORP as GLOBAL.
2. Link the STEPPS system and include module with GORP.

SAIL:

1. Define SGORP as INTERNAL and add 7 dummy parameters.
2. Add CALLSAIL (SGORP, GORP, 1); to file SETUP.BLI and recompile it.
3. Link the STEPPS system and include module with SGORP.

FORTRAN:

1. Define FGORP as the FORTRAN function.
2. Compile the following SAIL module:

```
ENTRY;
EXTERNAL FORTRAN PROCEDURE FGORP (ARRAY M);
INTERNAL PROCEDURE SGORP (ARRAY M; INTEGER D1,D2,D3,D4,D5,D6,D7);
 FGORP (M);
```

3. Do steps 2 and 3 for SAIL.

ATTRIBUTE

```
<ATTRIBUTE cmd>::= ATTRIBUTE <link name list> <link attributes>
<link attributes>  ::= <attribute assignment> | <attribute assignment> <link attributes>
<attribute assignment>  ::= QUEUE:<integer> | VOLUME:<integer> |
                DELAY:<real> | STARTUP:<real>
```

Each link named in the <link name list> is assigned the attributes named. For

example, to assign the attributes to link ALPHA of maximum queue length of 3 and delay time of 2.0, the following would work:

ATTRIBUTE ALPHA QUEUE:3 DELAY:2.0

An abbreviation allows several links to obtain the same attributes. Thus,

ATTRIBUTE ALPHA, BETA, GAMMA QUEUE:17, DELAY:4.0

assigns the same attributes to links ALPHA, BETA, and GAMMA.

CLEAR

<CLEAR cmd> ::= CLEAR <clear parameter>
<clear parameter>::= ALL | null | PROCESSES | LINKS

The result of this command is to clear the model of all PROCESSes, LINKs, or both. ***null*** is the same as ALL. For CLEAR ALL the model name is also reset to the default model name: MODEL.

COLLECT

<COLLECT cmd> ::= COLLECT <col key> <process name list>
<col key> ::= STATISTICS | NOSTATISTICS

The result of this command is to mark or unmark each process named for simulation statistics data collection. Each process in the <process name list> must already have been defined before issuing the command. The default is to COLLECT STATISTICS for each process.

CONTINUE

<CONTINUE cmd> ::= CONTINUE <time> <cont. param>
<cont. param> ::= TRACE | MODELTRACE |
FILETRACE <file> | <null>

This command is used to restart (or continue) a simulation where it halted (see SIMULATE). The <time> parameter is a real number representing the length of time the simulation should continue. TRACE means to display a simulation trace on the terminal

device. FILETRACE <file> extends the named PDP-10 file with the simulation trace. The extension TRA will always be used. MODELTRACE extends the file named <modelname>.TRA with the simulation trace. <modelname> is the current model name as set by the MODEL command.

COPY

The COPY command syntax has been changed since the examples in Chapter III were created. Both the old and new follow, although the new syntax is the actual syntax.

Old syntax

<COPY cmd>::= COPY <copy params> : <master item>

New syntax

<COPY cmd>::= COPY <master item> TO <copy params>

Common syntax

<copy params> ::= <link list> | <process list> | <port item list>
<port item>::= <port name> | <type-less port name> |
.<1 or 0><port number> | .<port number>

The purpose of the COPY command is to duplicate items to the left of the colon to have the same "attributes" as the item on the right of the colon. The actual semantics is based on the type of <master item> as follows:

LINKS For each named link, the attributes of the "master item" link are copied. Only the attributes are copied, but not any connections since ports can only be connected to one link.

Example

```
A.3←FOO
ATTRIBUTE FOO  QUEUE:3  VOLUME:7
COPY  FOO TO BAZ, GORP
```

Now BAZ and GORP are linked with identical attributes as FOO (Queue:3, Volume:7, Delay:0, and startup:0). However, neither is connected to any port even though FOO is connected to port A.13.

PROCESSES For each named process, all of the attributes of the <master item> process are duplicated. The attributes include connection to links, transition matrix, and simulation parameters. Only an unused name can be the result of COPY. Thus a process must be removed before its name can be used as a COPY of another process.

Example

```
A.3←FOO
A.01=*13:1.0/01:1
COPY A TO B,C
```

Assuming that B and C are previously unused names, they will now be identical to process A. Thus the above COPY command is a short cut for the following commands (assuming process A was previously undefined):

```
B.3←FOO
B.01=*13:1.0/01:1
C.3←FOO
C.01=*13:1.0/01:1
```

PORTS Named ports are copied based upon the <master item> port. The corresponding connections and transition matrix vector may be copied. If the master port does not exist, it will be created and similarly a new process may also be created. The ports named in the <port item list> may already exist. When a new process is created or the <port item> process has the same number of ports as the <master item> process, all probabilities and computation times associated with the <master item> port are set for the <port item port> When the above does not hold, a port only is created and given default properties.

The transition matrix values are set in the same order as the <master item> port; no examination is made for concurring port number. The repeating factor of the <port item> is also set to be the same as the <master item> port. When the ports, <port item> and <master item> are of the same type, of the same process, and not already connected to a link, then the <port item> port is connected to the same link as the <master item> port. COPY makes no change in a process's initial state since that is a process property, not a port property.

As a notational extension, the process name and/or port type may be left out of the <port item>. When this occurs, the previously named process or/and port type (to the left in the <port item list> is used. The initial default process name or/and port type is the <master item> port process name.

Example

```
FOO←A.2←BAZ
A.12=02:1/12:1
D.11 = 05:1/02:1
COPY A.12 TO A.14, D.02
```

The above copy command replaces the following commands (assuming process D did not exist previously):

```
A.14←BAZ
A.14 = 02:1
D.02 = 05:1   ! since 05 is the third port of D.
```

DENSITY

```
<DENSITY cmd>   ::= DENSITY <den. function type><den. params>
<den. function type>    ::= NORMAL | EXPONENTIAL
<den. param>    ::= PORT <port name> | LINK <link name>
                    FOR <positive integer> | GRAIN <positive integer>
                    MEAN <positive real> | VARIANCE <non-neg. real>
                    EPSILON <positive real>
```

Given a port name and a link name, connections and port transition values are generated to represent the named probability density function service rate as seen by the link to (or from) the port. The mean (default:10.0), variance (default:1.0), and appropriate grain (same as FOR; default:10) can be specified. EPSILON represents the density mass of the distribution tail(s) and is defaulted to 0.001.

An exact description of the result of the DENSITY command is as follows where the process name is PROCESS, given port type is ***TYPE*** and the given port number is ***n***.

1. Perform PROCESS.I***n***←PROCESS.O***n***, i.e. create a link and two ports.
2. Create ports PROCESS.***TYPEn***+1, . . ., PROCESS.***TYPEn***+(grain size) and connect them all to the named link.

3. Do PROCESS.On = In:1

4. Do PROCESS.*TYPE*n+1 = PROCESS.On:1

5. COPY PROCESS.*TYPE*n+1 TO PROCESS.*TYPE*n+2, . . ., PPOCESS.*TYPE*n+(grain size)

6. Set PROCESS.In transition matrix probabilities and time values dependent on the named density function. The successor ports are *TYPE*n+1, . . ., *TYPE*n+(grain size).

DISPLAY

<DISPLAY cmd> ::= DISPLAY <display arguments>
<display arguments> ::= ATTRIBUTES <link list> | COLLECT | COMPETE |
CONNECTIONS <port, process, link list> | DANGLING |
GRAPH <DISPLAY GRAPH parameters> | LINKS | LOOPS |
MODELNAME | PATHS <obj 1> TO <obj 2> | PORTS <process list> |
PRIORITY null | PRIORITY <process list> |
PROCESSES | SCHEDULER | TRANSITIONS <process, port list>
<DISPLAY GRAPH parameters> ::= ALL | ATTRIBUTES | JATTRIBUTES |
JCONNECTIONS | JTRANSITIONS | null

The DISPLAY command is used to display items in the STEPPS model and states of the STEPPS system (though not of a STEPPS simulation) on a terminal. Each argument is a command to display different objects and will be described below.

ATTRIBUTES Display the attributes of each link named.

COLLECT Display which processes will and will not collect statistics during a simulation.

COMPETE Display which processes will and will not compete for available processors during a simulation.

CONNECTIONS Display the connections to each port, process, and link named.

DANGLING Display which ports are not connected to any created link. These ports are connected to the special, non-createable link named DANGLING.

GRAPH ALL or GRAPH Display all link attributes and connections, all process transitions, all competing and non-competing processes, and all collecting and non-collecting processes.

GRAPH ATTRIBUTES Same as JATTRIBUTES *and* JCONNECTIONS.

GRAPH JATTRIBUTES Display just attributes of each link.

GRAPH JCONNECTIONS Display just the connections for the entire graph.

GRAPH JTRANSITIONS Display just the transitions of each process.

GRAPH TRANSITIONS Same as JTRANSITIONS *and* JCONNECTIONS.

LINKS Display the name of each link.

LOOPS Display each cycle in the graph.

MODELNAME Display the model name, the date, and the current time.

PATHS Display all paths between the named nodes.

PORTS Display the port names for each process named.

PRIORITY Display the priority number of each process named (or all).

PROCESSES Display the name of each process and its priority.

SCHEDULER Display the simulation scheduling discipline.

TRANSITIONS Display transitions for each port or entire process named.

EXIT

<EXIT cmd>::= EXIT

Exit from the STEPPS system. If the EXIT command is issued in a file that is LOADed, the result is to return to the STEPPS LOAD command (See LOAD).

LOAD

<LOAD cmd>::= LOAD <file name> <load param>
<load param> ::= ECHO | null

LOAD is used to retrieve STEPPS commands from a stored PDP-10 file. <file name> is the standard PDP-10 file name, viz. device:name.ext (only device DSKs are allowed). If no extension is used the extension TEP is assumed.

MODEL

<MODEL cmd> ::= MODEL <file name>

For convenience, each model can be named; default model name is MODEL. When a LOAD is performed, without any parameters, the model name becomes the LOAD file name. The MODEL command can be used at any time to change the current model name.

REMOVE

<REMOVE cmd> ::= REMOVE <port, process, link list>

Each item (port, process or link) in the parameter list is removed from the graph. When a link is removed, any port that had been connected to it becomes connected to the special link DANGLING. When a process has only one port, that port can not be removed; instead the process should be removed.

SAVE

<SAVE cmd>::= SAVE <save params>
<save param> ::= ALL | COMPETE | EXTEND | FILE <file name> |
GRAPH | LINKS | NODES | <list of nodes> |
PRIORITY | PROCESSES | SCHEDULER | null

The SAVE command is used to save a model description, components of a model description, and simulation parameters onto a PDP-10 file. Its common use is saving the entire description and parameters onto the file named by the MODEL command. This is accomplished by simply using SAVE with no parameters. Tha use of the parameter ALL is the same as ***null*** except that a file name is required. The format of the data written is the same as that used by the DISPLAY command, as follows:

ALL -- same as DISPLAY GRAPH ALL

COMPETE -- same as DISPLAY COMPETE

GRAPH -- same as DISPLAY GRAPH ATTRIBUTES ***and*** DISPLAY GRAPH JTRANSITIONS

LINKS -- same as DISPLAY GRAPH ATTRIBUTES

PRIORITY -- same as DISPLAY PRIORITY

PROCESSES -- same as DISPLAY GRAPH TRANSITIONS

SCHEDULER -- same as DISPLAY SCHEDULER

The FILE parameter is used to name the file to receive the data. No device can be specified.

The EXTEND parameter signifies that the named file is extended instead of replaced.

The NODES parameter signifies that the connections and attributes or transitions of the named nodes are to be saved.

SCHEDULE

```
<SCHEDULE cmd> ::= SCHEDULE <schedule parameter>
<schedule parameter>   ::= BY <scheduling style> |
                  COMPETE <processlist> | NONCOMPETE <process list> |
                  PRIORITY <process-priority list>
<process-priority>::= <process name>:<non-negative Integer>
<scheduling style>::= LINK | LKPR | PROCESS | PRLK | FIFO | RANDOM
```

The SCHEDULE command is used to set attributes for the simulation scheduler. The BY parameter is used to set the scheduling style. The PRIORITY parameter is used to set priorities for processes. The COMPETE and NONCOMPETE parameters are used to set which processes will and will not compete for available processors.

SIMULATE

```
<SIMULATE cmd> ::= SIMULATE <time> <sim params>
<time>      ::= <a non-negative real number>
<sim param>::= FILETRACE <file name> |
                  MODELTRACE | PROCESSORS <positive integer> |
                  SCHEDULE <scheduling style> | SEED <positive Integer> |
                  TRACE | WORKINGSET
```

The SIMULATE command is used to initiate a STEPPS model simulation for the length of time specified. The other parameters set simulation details as explained below. No more than one of the parameters TRACE, FILETRACE, and MODELTRACE can be used.

PROCESSORS sets the number of processors available for the simulation. SCHEDULE resets the simulation scheduling style: LINK, PROCESS, FIFO, RANDOM, LKPR, or PRLK. SEED sets the simulation random number generator seed for this simulation. Each simulation starts with the same internally defined seed unless specifically set by the SEED parameter.

The TRACE parameter causes messages to be displayed on the terminal describing each simulated event. FILETRACE extends the named file with the trace information. MODELTRACE extends the file <model name>.TRA with the trace information.

The WORKINGSET parameter causes those processes for which statistics are being collected to additionally collect statistics showing related working sets of processes.

SNAPSHOTS

<SNAPSHOTS cmd>::= SNAPSHOTS <snap params>
<snap param> ::= FILE <optional file name> | LINKS |
PROCESSES | NODES <process, link list> | SCHEDULER | null

The SNAPSHOTS command is used to display current status of a simulation that has stopped, but not been terminated (UNSIMULATE).

The FILE parameter designates that the snapshot is to extend the file named (colon precedes the file name) or the <model name>.TRA file. The other parameters name the items to be examined; namely the SCHEDULER, all LINKS, all PROCESSES, or individually named nodes. A ***null*** parameter means all items.

STATISTICS

<STATISTICS cmd>::= STATISTICS <stat params>
<stat param> ::= FILE <optional file name> | LINKS |
PROCESSES | NODES <process, link list> | SCHEDULER | null

The STATISTICS command is used to display the current accumulated statistics of a simulation that has been stopped, but not terminated (UNSIMULATEd).

The meanings of the parameters are the same as for the SNAPSHOT command.

TEST

```
<TEST cmd>::= TEST <test param>
<test param>      ::= GRAPH | DEADLOCK <test dead param> |
                      NODES <process list>
<test dead param>::= TRACE | VERBOSE | NSAVE |
                      NSTRACE | NSVERBOSE
```

The TRACE command is used to analyze the structure of a STEPPS model. The GRAPH parameter means to determine whether the entire graph is well-formed (including each process). The NODES parameter is used to determine whether individual processes are well-formed.

The DEADLOCK parameter means to determine whether any deadlocks exist in a STEPPS model. The process destroys the model, so an automatic SAVE is normally performed to a unique file before the deadlock test procedure begins and the model is normally restored afterwards. Two types of traces can be performed showing how the deadlock algorithm works. The DEADLOCK subparameters are used to determine how the saves and tracer are performed.

TRACE -- Trace the application of each reduction.

VERBOSE -- Same as TRACE plus display all transition matrix changes.

NSAVE -- Allow the model to be destroyed without being saved first nor restored afterwards.

NSTRACE -- NSAVE + TRACE.

NSVERBOSE -- NSAVE + VERBOSE.

UNSIMULATE

```
<UNSIMULATE cmd>      ::= UNSIMULATE
```

The UNSIMULATE command is used to terminate a simulation that has stopped so that it can not be restarted (CONTINUEd).

APPENDIX B

Using the STEPPS System

This appendix presents a protocol of the use of the STEPPS system for the Chapter III Bliss/11 example. A discussion of the Chapter III Hearsay II example input problem and its solution is also presented.

B.1. Bliss/11 example protocol

An annotated protocol of the use of the STEPPS system for the Bliss/11 model shown in Figure III-11 is presented below. Following the protocol, the simulation commands used for the experiments will be presented. A sample of the statistics produced upon request after a simulation will also be presented.

```
! PROTOCOL FOR BLISS/11
*MODEL B11

*DENSITY EXPON PORT LEX.00 LINK LS MEAN .26
Port LEX.0000 Link LS Mean  .26000 Epsilon  .00100 For (Grain) 010
Link name "LINK001" will be used.

*DENSITY EXPON PORT SYNFLO.00 LINK SO MEAN .216
Port SYNFLO.0000 Link SO Mean  .21600 Epsilon  .00100 For (Grain) 010
Link name "LINK002" will be used.
*SYNFLO.I20=LS                  ! INPUT FROM "LEX"
*SYNFLO.00= I0:0; I20:1         ! AFTER OUTPUT, INPUT FROM "LEX"
*SYNFLO.I20= I0:1              ! REQUEST MORE INPUT

*DENSITY EXPON PORT DELAY.00 LINK OT MEAN 0.037
Port DELAY.0000 Link OT Mean  .03700 Epsilon  .00100 For (Grain) 010
Link name "LINK003" will be used.
*DELAY.I20=SO                   ! INPUT FROM "SYNFLO"

*DENSITY EXPON PORT TNBIND.00 LINK TC MEAN .122
Port TNBIND.0000 Link TC Mean  .12200 Epsilon  .00100 For (Grain) 010
Link name "LINK004" will be used.
*TNBIND.I20=OT                 ! INPUT FROM "DELAY"

*DENSITY EXPON PORT CODE.00 LINK CF MEAN .084
Port CODE.0000 Link CF Mean  .08400 Epsilon  .00100 For (Grain) 010
```

```
Link name "LINK005" will be used.
*CODE.I20=TC                    ! INPUT FROM "TNBIND"

*DENSITY EXPON PORT FINAL.D0 LINK FP MEAN .296
Port FINAL.D000 Link FR Mean  .29600 Epsilon  .00100 For (Grain) 010
Link name "LINK006" will be used.
*FINAL.I20=CF                   ! INPUT FROM "CODE"

*COPY DELAY.I20, TNBIND.I20, CODE.I20, FINAL.I20 : SYNFLO.I20
*COPY DELAY.DC, TNBIND.D0, CODE.D0, FINAL.D0 : SYNFLO.D0

*RESULT.I0=FR                   ! DEPOSITORY FOR RESULTS

*SCHEDULE NONCOMPETE RESULT

*ATTRIBUTE TC,CF,DT,FR,LS,SD  QUEUE: 10
```

The simulations were initiated by using the SIMULATE command. The following command was used to simulate the model using 6 processors and the FIFO scheduling algorithm for 100 time units:

```
Simulate 100 processors 6 schedule fifo
```

The other Bliss/11 experiments were simulated by modifying the SIMULATE command parameters for timing, number of processors, and scheduling algorithms as described in Appendix A. In order to eliminate the requirement for recreating the model for each simulation, the model was first written on a file (using the SAVE command) and for each simulation it was restored (using the LOAD command). A sample of the statistics displayed for links is shown below:

```
*statistics   ! at time = 100
Model B11 ! 19-May-76 03:46
************************************************************
Statistics at time   100.000    6 processors.

Link        Time      No.   No.   No.    Ex.     Ex.     Ex.    % Time % No.   % Time  % Time  Access  Request Sends
            Inactive Start Sends Posts   Qlen    Wait    Ovflo  Inactive Startups Startup Active  Rate    Rate    Rate
CF           100.00     1   337   328  9.175    .807    .574  100.00%    .15%    .00%    .00%    .15     .30     .30
DT           100.00     1   360   349  4.529    .287    .278  100.00%    .14%    .00%    .00%    .14     .29     .28
FP           100.00   328   327   328   .000   1.000    .000  100.00%  50.08%    .00%    .00%    .15     .30     .31
LINK001      100.00   378   377   377   .000    .000    .000  100.00%  50.13%    .00%    .00%    .13     .26     .26
LINK002      100.00   324   371   371   .128    .000    .000  100.00%  43.67%    .00%    .00%    .13     .27     .27
LINK003      100.00   128   360   360   .578    .000    .000  100.00%  17.78%    .00%    .00%    .14     .28     .28
LINK004      100.00   237   349   349   .287    .000    .000  100.00%  33.95%    .00%    .00%    .14     .29     .29
LINK005      100.00   277   338   338   .139    .000    .000  100.00%  40.96%    .00%    .00%    .15     .30     .30
LINK006      100.00   328   328   328   .007    .000    .000  100.00%  50.00%    .00%    .00%    .15     .30     .30
LS           100.00     2   376   371  3.915    .128    .036  100.00%    .27%    .00%    .00%    .13     .27     .26
SD           100.00     2   371   360  1.862    .578    .027  100.00%    .27%    .00%    .00%    .14     .28     .27
TC           100.00     2   349   338  6.226    .139    .298  100.00%    .29%    .00%    .00%    .15     .30     .29
```

B.2. The STEPPS Hearsay II model

The STEPPS model of the Hearsay II system was discussed in Chapter III. However, unlike the Bliss/11 model, the exact Hearsay II model was not shown since it is too large to place into the text of Chapter III. It was found that many of the structures used for the Hearsay II model were similar, but not close enough to utilize the STEPPS command COPY to facilitate input of the model. A STEPPS feature discussed in Chapter VI as a future system tool for reproducing groups of processes and links might have been useful. Instead of implementing that feature, the action pursued was to create a simple preprocessor program (in SAIL) to convert a description of a Hearsay II model into a form appropriate to the STEPPS system. The following is an example of the inout to the preprocessor. The actual probabilities used in the STEPPS Hearsay II model are shown.

```
process hslelo locks c1,c2,c5,word,phon
compute 1100 done

process prelelo locks c11,c1?e,c11b,c11e,c11f,c11g,c11h,c14,c14e,c14b
compute 0.70 invoke hslelo ;

process prelpsyn locks cpseg,pseg 1,pseg 2
compute 50
invoke hslcseg:.02 650
invoke hslpsyn:.951 650
done

process hslpsyn
locks c9,c9e,c9f,c9g,c9h,c10,c10e,c10b,c12,
   c12b,c12e,c12f,c12g,c12h,c13,c13e,c13b,cmxn,mxn 1,mxn 2,cpseg,pseg 1,pseg 2
compute 25.850
done

process hslcseg
locks c9,c9e,c9f,c9g,c9h,c10,c10e,c10b,c12,
   c12c,c12e,c12f,c12g,c12h,c13,c13e,c13b,cmxn,mxn 1,mxn 2,cpseg,pseg 1,pseg 2
compute 25.445
done

process prelrpol
locks shdsent,shdword,word,wrdsurn surn,phon,cmxn,mxn 1,mxn 2,cpseg,pseg 1,pseg 2,seg
compute 31
invoke hsluv:.376
done

process hsluv
locks c1,c2,c4,c5,c7,c7e,c7b,c8,c9,c9e,c9f,c9g,c9h,c10,
   c10e,c10b,c11,c11e,c11b,c11e,c11f,c11g,c11h,c12,
   c12c,c12e,c12f,c12g,c12h,c14,c14e,c14b
compute 134
done
```

```
process preiseg
locks cpseg,pseg 1,pseg 2,seg
compute 35,100
invoke ksiseg
done

process ksiseg
locks seg
compute 400
compute 400
compute 400
compute 400
done

process preiutb
locks c4,c5
compute 50
invoke ksiutb: 057 60
done

process ksiutb
locks c2,c3,c3e,c3b,c4,c5,c6,c6e,c6f,c6g,c6h,c13,c13e,c13b
compute 30
compute 30,150
compute 30,150
done

process preipsc
locks c3,c3e,c3b,c7,c7e,c7b
compute 50
invoke ksisearch:,13 135,ksitime:,13 135
done

process ksisearch
locks c2,c3,c3e,c3b,c4,c5,c6,c6e,c6f,c6g,c6h,c7,c7e,c7b,c8,c13,c13e,c13b
compute 50,1100
compute 200,1100
done

process ksitime
locks c2,c3,c3e,c3b,c4,c5,c6,c6e,c6f,c6g,c6h,c7,c7e,c7b,c8,c13,c13e,c13b
compute 50,225
compute 75,225
done
levicon seg,pseg 1,pseg 2,mxn 1,mxn 2,phon,surn,wrdsurn,word,
      shdword,shdsent

crlocks

cpseg pseg 2 pseg 1
cmxn mxn 2 mxn 1
c2 c1 surn
c3 word c7
c4 wrdsurn c8
c5 wrdsurn surn
c6 c8 c12
c7 surn c10
c8 surn phon
c9 phon c12
c10 phon cmxn
c11 cmxn c14
c12 cmxn cpseg
c13 phon cpseg
c14 cpseg seg
c14a pseg 1 seg
c14b pseg 2 seg
c13a phon pseg 1
c13b phon pseg 2

c12e mxn 1 pseg 1
c12f mxn 1 pseg 2
c12g mxn 2 pseg 1
c12h mxn 2 pseg 2

c11a mxn 1 c14
c11b mxn 2 c14
```

B.2 The STEPPS Hearsay II model

```
c11e mxn 1 c14e
c11f mxn 1 c14b
c11g mxn 2 c14e
c11h mxn 2 c14b

c10e phon mxn 1
c10b phon mxn 2

c9e phon c12e
c9f phon c12f
c9g phon c12g
c9h phon c12h

c7e surn c10e
c7b surn c10b

c6e c8 c12e
c6f c8 c12f
c6g c8 c12g
c6h c8 c12h

c3e word c7e
c3b word c7b

done

fini
```

The result of the Hearsay II model generation is the STEPPS model which follows:

```
model nwlk1 ' locking by the 7 ls*icon levels + 2 sublevels
'                Toprel:pol      Toprelpsc      Toprelpsyn
pcselector.:0= o1:.75/o0:1:      o2:0.121/o0:1:   o3: 0.052/o0:1:
'                Toprelseg       TopreIUtlb     TopreIAlo
Pcselector.:0=  o4:.009/o0:1:      o5:0.006/o0:1:   o6: 0.002/o0:1:
pcselector.o0=:0:1:1    ' Messeges come every 1 unit of time
pcselector.0=>pcsel>pcselector.0
prelrpol.1>toprelrpol>pcselector.1
prelpsc.1>toprelpsc>pcselector.2
prelpsyn.1>toprelpsyn>pcselector.3
prelseg.1>toprelseg>pcselector.4
prelutb.1>toprelutb>pcselector.5
prelelo.1>toprelelo>pcselector.6
sched noncompete pcselector


' kslelo Locks c1,c2,c5,word,phon
kslelo.:1=* o101: .200
kslelo.:1= o102: .200
kslelo.:1= o103: .200
kslelo.:1= o104: .200
kslelo.:1= o105: .200
twlic1>kslelo.101>fwllc1
twulc1>kslelo.301
kslelo.o101=:101:1/o301: 1, 1100.000/o2:1
twllc2>kslelo.102>fwllc2
twulc2>kslelo.302
kslelo.o102=:102:1/o302: 1, 1100.000/o2:1
twllc5>kslelo.103>fwllc5
twulc5>kslelo.303
kslelo.o103=:103:1/o303: 1, 1100.000/o2:1
tlklword>kslelo.104>flklword
ulklword>kslelo.304
kslelo.o104=:104:1/o304: 1, 1100.000/o2:1
tlklphon>kslelo.105>flklphon
ulklphon>kslelo.305
kslelo.o105=:105:1/o305: 1, 1100.000/o2:1
kslelo.:2>kslelo.o2
kslelo.o2=:2:1
kslelo.:2= :1:1

' prelelo Locks c11,c11e,c11b,c11a,c11f,c11g
'              ,c11h,c14,c14e,c14b
prelelo.:1=* o101: .100
prelelo.:1= o102: .100
prelelo.:1= o103: .100
prelelo.:1= o104: .100
prelelo.:1= o105: .100
prelelo.:1= o106: .100
prelelo.:1= o107: .100
prelelo.:1= o108: .100
prelelo.:1= o109: .100
prelelo.:1= o110: .100
twllc11>prelelo.101>fwllc11
twulc11>prelelo.301
prelelo.o101=:101:1/o301: 1, 70.000/o2:1
twllc11e>prelelo.102>fwllc11e
twulc11e>prelelo.302
prelelo.o102=:102:1/o302: 1, 70.000/o2:1
twllc11b>prelelo.103>fwllc11b
twulc11b>prelelo.303
prelelo.o103=:103:1/o303: 1, 70.000/o2:1
twllc11e>prelelo.104>fwllc11e
twulc11e>prelelo.304
prelelo.o104=:104:1/o304: 1, 70.000/o2:1
twllc11f>prelelo.105>fwllc11f
twulc11f>prelelo.305
prelelo.o105=:105:1/o305: 1, 70.000/o2:1
twllc11g>prelelo.106>fwllc11g
twulc11g>prelelo.306
prelelo.o106=:106:1/o306: 1, 70.000/o2:1
twllc11h>prelelo.107>fwllc11h
twulc11h>prelelo.307
prelelo.o107=:107:1/o307: 1, 70.000/o2:1
twllc14>prelelo.108>fwllc14
twulc14>prelelo.308
prelelo.o108=:108:1/o308: 1, 70.000/o2:1
twllc14e>prelelo.109>fwllc14e
twulc14e>prelelo.309
prelelo.o109=:109:1/o309: 1, 70.000/o2:1
twllc14b>prelelo.110>fwllc14b
twulc14b>prelelo.310
prelelo.o110=:110:1/o310: 1, 70.000/o2:1
prelelo.:2>prelelo.o2
prelelo.o2=:2:1
kslelo.1>vblkslelo>prelelo.901
prelelo.:2=o901:1/o3:1
prelelo.:3>prelelo.o3
prelelo.o3=:3:1
prelelo.:3= :1:1

' prelpsyn Locks cpseg.pseg 1,pseg 2
prelpsyn.:1=* o101: .333
prelpsyn.:1= o102: .333
prelpsyn.:1= o103: .334
twllcpseg>prelpsyn.101>fwllcpseg
twulcpseg>prelpsyn.301
prelpsyn.o101=:101:1/o301: 1, 50.000/o2:1
tlklpseg 1>prelpsyn.102>flklpseg 1
ulklpseg 1>prelpsyn.302
prelpsyn.o102=:102:1/o302: 1, 50.000/o2:1
tlklpseg 2>prelpsyn.103>flklpseg 2
ulklpseg 2>prelpsyn.303
prelpsyn.o103=:103:1/o303: 1, 50.000/o2:1
prelpsyn.:2>prelpsyn.o2
prelpsyn.o2=:2:1
kslcseg.1>vblkslcseg>prelpsyn.901
prelpsyn.:2=o901: .902,650/o3:1
prelpsyn.:2=o3: .098,650
prelpsyn.:3>prelpsyn.o3
prelpsyn.o3=:3:1
kslpsyn.1>vblkslpsyn>prelpsyn.902
prelpsyn.:3=o902: .951,650/o4:1
prelpsyn.:3=o4: .049,650
prelpsyn.:4>prelpsyn.o4
prelpsyn.o4=:4:1
prelpsyn.:4= :1:1

' kslpsyn Locks c9,c9e,c9f,c9a,c9h,c10,c10e,
'      c10b,c12,c12b,c12e,c12f,c12g,c12h,c13,
'      c13e,c13b,cmxn,mxn 1,mxn 2,cpseg,
'      pseg 1,pseg 2
kslpsyn.:1=* o101: .043, 25.000
kslpsyn.:1= o102: .043, 25.000
kslpsyn.:1= o103: .043, 25.000
kslpsyn.:1= o104: .043, 25.000
kslpsyn.:1= o105: .043, 25.000
kslpsyn.:1= o106: .043, 25.000
kslpsyn.:1= o107: .043, 25.000
kslpsyn.:1= o108: .043, 25.000
kslpsyn.:1= o109: .043, 25.000
kslpsyn.:1= o110: .043, 25.000
kslpsyn.:1= o111: .043, 25.000
kslpsyn.:1= o112: .043, 25.000
kslpsyn.:1= o113: .044, 25.000
kslpsyn.:1= o114: .044, 25.000
kslpsyn.:1= o115: .044, 25.000
kslpsyn.:1= o116: .044, 25.000
kslpsyn.:1= o117: .044, 25.000
kslpsyn.:1= o118: .044, 25.000
kslpsyn.:1= o119: .044, 25.000
kslpsyn.:1= o120: .044, 25.000
kslpsyn.:1= o121: .044, 25.000
kslpsyn.:1= o122: .044, 25.000
kslpsyn.:1= o123: .044, 25.000
twllc9>kslpsyn.101>fwllc9
twulc9>kslpsyn.301
kslpsyn.o101=:101:1/o301: 1, 850.000/o2:1
twllc9e>kslpsyn.102>fwllc9e
twulc9e>kslpsyn.302
kslpsyn.o102=:102:1/o302: 1, 850.000/o2:1
twllc9f>kslpsyn.103>fwllc9f
twulc9f>kslpsyn.303
kslpsyn.o103=:103:1/o303: 1, 850.000/o2:1
twllc9a>kslpsyn.104>fwllc9g
twulc9a>kslpsyn.304
kslpsyn.o104=:104:1/o304: 1, 850.000/o2:1
twllc9h>kslpsyn.105>fwllc9h
twulc9h>kslpsyn.305
kslpsyn.o105=:105:1/o305: 1, 850.000/o2:1
twllc10>kslpsyn.106>fwllc10
```

```
twulc10-kslpsyn.306
kslpsyn.o106=:106:1/o306: 1, 850.000/o2:1
twllc10a-kslpsyn.107-fwllc10a
twulc10a-kslpsyn.307
kslpsyn.o107=:107:1/o307: 1, 850.000/o2:1
twllc10b-kslpsyn.108-fwllc10b
twulc10b-kslpsyn.308
kslpsyn.o108=:108:1/o308: 1, 850.000/o2:1
twllc12-kslpsyn.109-fwllc12
twulc12-kslpsyn.309
kslpsyn.o109=:109:1/o309: 1, 850.000/o2:1
twllc12b-kslpsyn.110-fwllc12b
twulc12b-kslpsyn.310
kslpsyn.o110=:110:1/o310: 1, 850.000/o2:1
twllc12e-kslpsyn.111-fwllc12e
twulc12e-kslpsyn.311
kslpsyn.o111=:111:1/o311: 1, 850.000/o2:1
twllc12f-kslpsyn.112-fwllc12f
twulc12f-kslpsyn.312
kslpsyn.o112=:112:1/o312: 1, 850.000/o2:1
twllc12g-kslpsyn.113-fwllc12g
twulc12g-kslpsyn.313
kslpsyn.o113=:113:1/o313: 1, 850.000/o2:1
twllc12h-kslpsyn.114-fwllc12h
twulc12h-kslpsyn.314
kslpsyn.o114=:114:1/o314: 1, 850.000/o2:1
twllc13-kslpsyn.115-fwllc13
twulc13-kslpsyn.315
kslpsyn.o115=:115:1/o315: 1, 850.000/o2:1
twllc13e-kslpsyn.116-fwllc13e
twulc13e-kslpsyn.316
kslpsyn.o116=:116:1/o316: 1, 850.000/o2:1
twllc13b-kslpsyn.117-fwllc13b
twulc13b-kslpsyn.317
kslpsyn.o117=:117:1/o317: 1, 850.000/o2:1
twllcmxn-kslpsyn.118-fwllcmxn
twulcmxn-kslpsyn.318
kslpsyn.o118=:118:1/o318: 1, 850.000/o2:1
tlklmxn 1-kslpsyn.119-flklmxn 1
ulklmxn 1-kslpsyn.319
kslpsyn.o119=:119:1/o319: 1, 850.000/o2:1
tlklmxn 2-kslpsyn.120-flklmxn 2
ulklmxn 2-kslpsyn.320
kslpsyn.o120=:120:1/o320: 1, 850.000/o2:1
twllcpseg-kslpsyn.121-fwllcpseg
twulcpseg-kslpsyn.321
kslpsyn.o121=:121:1/o321: 1, 850.000/o2:1
tlklpseg 1-kslpsyn.122-flklpseg 1
ulklpseg 1-kslpsyn.322
kslpsyn.o122=:122:1/o322: 1, 850.000/o2:1
tlklpseg 2-kslpsyn.123-flklpseg 2
ulklpseg 2-kslpsyn.323
kslpsyn.o123=:123:1/o323: 1, 850.000/o2:1
kslpsyn.i2-kslpsyn.o2
kslpsyn.o2=:2:1
kslpsyn.i2= :1:1

! kslcseg Locks c9,c9e,c9f,c9g,c9h,c10,
!       c10e,c10b,c12,c12c,c12e,c12f,c12g,
!       c12h,c13,c13e,c13b,cmxn,mxn 1,mxn 2,
!       cpseg,pseg 1,pseg 2
kslcseg.:1=* o101: .043, 25.000
kslcseg.:1= o102: .043, 25.000
kslcseg.:1= o103: .043, 25.000
kslcseg.:1= o104: .043, 25.000
kslcseg.:1= o105: .043, 25.000
kslcseg.:1= o106: .043, 25.000
kslcseg.:1= o107: .043, 25.000
kslcseg.:1= o108: .043, 25.000
kslcseg.:1= o109: .043, 25.000
kslcseg.:1= o110: .043, 25.000
kslcseg.:1= o111: .043, 25.000
kslcseg.:1= o112: .043, 25.000
kslcseg.:1= o113: .044, 25.000
kslcseg.:1= o114: .044, 25.000
kslcseg.:1= o115: .044, 25.000
kslcseg.:1= o116: .044, 25.000
kslcseg.:1= o117: .044, 25.000
kslcseg.:1= o118: .044, 25.000
kslcseg.:1= o119: .044, 25.000
kslcseg.:1= o120: .044, 25.000
kslcseg.:1= o121: .044, 25.000
kslcseg.:1= o122: .044, 25.000
kslcseg.:1= o123: .044, 25.000
twllc9-kslcseg.101-fwllc9
twulc9-kslcseg.301
kslcseg.o101=:101:1/o301: 1, 445.000/o2:1
twllc9e-kslcseg.102-fwllc9e
twulc9e-kslcseg.302
kslcseg.o102=:102:1/o302: 1, 445.000/o2:1
twllc9f-kslcseg.103-fwllc9f
twulc9f-kslcseg.303
kslcseg.o103=:103:1/o303: 1, 445.000/o2:1
twllc9g-kslcseg.104-fwllc9g
twulc9g-kslcseg.304
kslcseg.o104=:104:1/o304: 1, 445.000/o2:1
twllc9h-kslcseg.105-fwllc9h
twulc9h-kslcseg.305
kslcseg.o105=:105:1/o305: 1, 445.000/o2:1
twllc10-kslcseg.106-fwllc10
twulc10-kslcseg.306
kslcseg.o106=:106:1/o306: 1, 445.000/o2:1
twllc10a-kslcseg.107-fwllc10a
twulc10a-kslcseg.307
kslcseg.o107=:107:1/o307: 1, 445.000/o2:1
twllc10b-kslcseg.108-fwllc10b
twulc10b-kslcseg.308
kslcseg.o108=:108:1/o308: 1, 445.000/o2:1
twllc12-kslcseg.109-fwllc12
twulc12-kslcseg.309
kslcseg.o109=:109:1/o309: 1, 445.000/o2:1
twllc12c-kslcseg.110-fwllc12c
twulc12c-kslcseg.310
kslcseg.o110=:110:1/o310: 1, 445.000/o2:1
twllc12e-kslcseg.111-fwllc12e
twulc12e-kslcseg.311
kslcseg.o111=:111:1/o311: 1, 445.000/o2:1
twllc12f-kslcseg.112-fwllc12f
twulc12f-kslcseg.312
kslcseg.o112=:112:1/o312: 1, 445.000/o2:1
twllc12g-kslcseg.113-fwllc12g
twulc12g-kslcseg.313
kslcseg.o113=:113:1/o313: 1, 445.000/o2:1
twllc12h-kslcseg.114-fwllc12h
twulc12h-kslcseg.314
kslcseg.o114=:114:1/o314: 1, 445.000/o2:1
twllc13-kslcseg.115-fwllc13
twulc13-kslcseg.315
kslcseg.o115=:115:1/o315: 1, 445.000/o2:1
twllc13e-kslcseg.116-fwllc13e
twulc13e-kslcseg.316
kslcseg.o116=:116:1/o316: 1, 445.000/o2:1
twllc13b-kslcseg.117-fwllc13b
twulc13b-kslcseg.317
kslcseg.o117=:117:1/o317: 1, 445.000/o2:1
twllcmxn-kslcseg.118-fwllcmxn
twulcmxn-kslcseg.318
kslcseg.o118=:118:1/o318: 1, 445.000/o2:1
tlklmxn 1-kslcseg.119-flklmxn 1
ulklmxn 1-kslcseg.319
kslcseg.o119=:119:1/o319: 1, 445.000/o2:1
tlklmxn 2-kslcseg.120-flklmxn 2
ulklmxn 2-kslcseg.320
kslcseg.o120=:120:1/o320: 1, 445.000/o2:1
twllcpseg-kslcseg.121-fwllcpseg
twulcpseg-kslcseg.321
kslcseg.o121=:121:1/o321: 1, 445.000/o2:1
tlklpseg 1-kslcseg.122-flklpseg 1
ulklpseg 1-kslcseg.322
kslcseg.o122=:122:1/o322: 1, 445.000/o2:1
tlklpseg 2-kslcseg.123-flklpseg 2
ulklpseg 2-kslcseg.323
kslcseg.o123=:123:1/o323: 1, 445.000/o2:1
kslcseg.i2-kslcseg.o2
kslcseg.o2=:2:1
kslcseg.i2= :1:1

! prelrpol Locks shdsent,shdword,word,
!       wrdsurn,surn,phon,cmxn,mxn 1,mxn 2
!       ,cpseg,pseg 1,pseg 2,seg
prelrpol.:1=* o101: .076
prelrpol.:1= o102: .077
prelrpol.:1= o103: .077
prelrpol.:1= o104: .077
```

```
prelrpol.r1* o105: .077
prelrpol.r1* o106: .077
prelrpol.r1* o107: .077
prelrpol.r1* o108: .077
prelrpol.r1* o109: .077
prelrpol.r1* o110: .077
prelrpol.r1* o111: .077
prelrpol.r1* o112: .077
prelrpol.r1* o113: .077
tlk|shdsent*prelrpol.101*flk|shdsent
ulk|shdsent*prelrpol.301
prelrpol.o101*r101:1/o301: 1, 31.000/o2:1
tlk|shdword*prelrpol.102*flk|shdword
ulk|shdword*prelrpol.302
prelrpol.o102*r102:1/o302: 1, 31.000/o2:1
tlk|word*prelrpol.103*flk|word
ulk|word*prelrpol.303
prelrpol.o103*r103:1/o303: 1, 31.000/o2:1
tlk|wrdsurn*prelrpol.104*flk|wrdsurn
ulk|wrdsurn*prelrpol.304
prelrpol.o104*r104:1/o304: 1, 31.000/o2:1
tlk|surn*prelrpol.105*flk|surn
ulk|surn*prelrpol.305
prelrpol.o105*r105:1/o305: 1, 31.000/o2:1
tlk|phon*prelrpol.106*flk|phon
ulk|phon*prelrpol.306
prelrpol.o106*r106:1/o306: 1, 31.000/o2:1
twl|cmvn*prelrpol.107*fwl|cmvn
twu|cmvn*prelrpol.307
prelrpol.o107*r107:1/o307: 1, 31.000/o2:1
tlk|mvn 1*prelrpol.108*flk|mvn 1
ulk|mvn 1*prelrpol.308
prelrpol.o108*r108:1/o308: 1, 31.000/o2:1
tlk|mvn 2*prelrpol.109*flk|mvn 2
ulk|mvn 2*prelrpol.309
prelrpol.o109*r109:1/o309: 1, 31.000/o2:1
twl|cpseg*prelrpol.110*fwl|cpseg
twu|cpseg*prelrpol.310
prelrpol.o110*r110:1/o310: 1, 31.000/o2:
tlk|pseg 1*prelrpol.111*flk|pseg 1
ulk|pseg 1*prelrpol.311
prelrpol.o111*r111:1/o311: 1, 31.000/o2:1
tlk|pseg 2*prelrpol.112*flk|pseg 2
ulk|pseg 2*prelrpol.312
prelrpol.o112*r112:1/o312: 1, 31.000/o2:1
tlk|seg*prelrpol.113*flk|seg
ulk|seg*prelrpol.313
prelrpol.o113*r113:1/o313: 1, 31.000/o2:1
prelrpol.r2*prelrpol.o2
prelrpol.o2*r2:1
ksluv 1*vk|ksluv*prelrpol.901
prelrpol.r2*o901: .376/o3:1
prelrpol.r2*o3: .624
prelrpol.r3*prelrpol.o3
prelrpol.o3*r3:1
prelrpol.r3* r1:1

! keluv Locke c1,c2,c4,c5,c7,c7a,c7b,c8,
!       c9,c9a,c9f,c9g,c9h,c10,c10a,c10b,
!       c11,c11e,c11b,c11e,c11f,c11g,c11h,
!       c12,c12c,c12e,c12f,c12g,c12h,c14,
!       c14e,c14b
ksluv.r1** o101: .031
keluv.r1* o102: .031
ksluv.r1* o103: .031
ksluv.r1* o104: .031
ksluv.r1* o105: .031
keluv.r1* o106: .031
ksluv.r1* o107: .031
ksluv.r1* o108: .031
ksluv.r1* o109: .031
ksluv.r1* o110: .031
keluv.r1* o111: .031
ksluv.r1* o112: .031
keluv.r1* o113: .031
ksluv.r1* o114: .031
ksluv.r1* o115: .031
keluv.r1* o116: .031
keluv.r1* o117: .031
ksluv.r1* o118: .031
ksluv.r1* o119: .031
keluv.r1* o120: .031
ksluv.r1* o121: .031
ksluv.r1* o122: .031
ksluv.r1* o123: .031
ksluv.r1* o124: .031
ksluv.r1* o125: .032
ksluv.r1* o126: .032
ksluv.r1* o127: .032
ksluv.r1* o128: .032
ksluv.r1* o129: .032
ksluv.r1* o130: .032
ksluv.r1* o131: .032
ksluv.r1* o132: .032
twl|c1*ksluv.101*fwl|c1
twu|c1*ksluv.301
keluv.o101*r101:1/o301: 1, 134.000/o2:1
twl|c2*ksluv.102*fwl|c2
twu|c2*ksluv.302
ksluv.o102*r102:1/o302: 1, 134.000/o2:1
twl|c4*ksluv.103*fwl|c4
twu|c4*ksluv.303
ksluv.o103*r103:1/o303: 1, 134.000/o2:1
twl|c5*ksluv.104*fwl|c5
twu|c5*ksluv.304
ksluv.o104*r104:1/o304: 1, 134.000/o2:1
twl|c7*ksluv.105*fwl|c7
twu|c7*ksluv.305
ksluv.o105*r105:1/o305: 1, 134.000/o2:1
twl|c7a*ksluv.106*fwl|c7a
twu|c7a*ksluv.306
ksluv.o106*r106:1/o306: 1, 134.000/o2:1
twl|c7b*keluv.107*fwl|c7b
twu|c7b*ksluv.307
ksluv.o107*r107:1/o307: 1, 134.000/o2:1
twl|c8*keluv.108*fwl|c8
twu|c8*ksluv.308
ksluv.o108*r108:1/o308: 1, 134.000/o2:1
twl|c9*ksluv.109*fwl|c9
twu|c9*ksluv.309
ksluv.o109*r109:1/o309: 1, 134.000/o2:1
twl|c9a*ksluv.110*fwl|c9a
twu|c9a*ksluv.310
keluv.o110*r110:1/o310: 1, 134.000/o2:1
twl|c9f*ksluv.111*fwl|c9f
twu|c9f*ksluv.311
ksluv.o111*r111:1/o311: 1, 134.000/o2:1
twl|c9g*ksluv.112*fwl|c9g
twu|c9g*ksluv.312
ksluv.o112*r112:1/o312: 1, 134.000/o2:1
twl|c9h*ksluv.113*fwl|c9h
twu|c9h*ksluv.313
keluv.o113*r113:1/o313: 1, 134.000/o2:1
twl|c10*ksluv.114*fwl|c10
twu|c10*ksluv.314
ksluv.o114*r114:1/o314: 1, 134.000/o2:1
twl|c10a*ksluv.115*fwl|c10a
twu|c10a*keluv.315
ksluv.o115*r115:1/o315: 1, 134.000/o2:1
twl|c10b*keluv.116*fwl|c10b
twu|c10b*ksluv.316
ksluv.o116*r116:1/o316: 1, 134.000/o2:1
twl|c11*ksluv.117*fwl|c11
twu|c11*ksluv.317
keluv.o117*r117:1/o317: 1, 134.000/o2:1
twl|c11a*ksluv.118*fwl|c11a
twu|c11a*ksluv.318
ksluv.o118*r118:1/o318: 1, 134.000/o2:1
twl|c11b*keluv.119*fwl|c11b
twu|c11b*ksluv.319
keluv.o119*r119:1/o319: 1, 134.000/o2:1
twl|c11e*ksluv.120*fwl|c11e
twu|c11e*ksluv.320
keluv.o120*r120:1/o320: 1, 134.000/o2:1
twl|c11f*ksluv.121*fwl|c11f
twu|c11f*ksluv.321
keluv.o121*r121:1/o321: 1, 134.000/o2:1
twl|c11g*ksluv.122*fwl|c11g
twu|c11g*ksluv.322
ksluv.o122*r122:1/o322: 1, 134.000/o2:1
twl|c11h*keluv.123*fwl|c11h
twu|c11h*keluv.323
ksluv.o123*r123:1/o323: 1, 134.000/o2:1
twl|c12*keluv.124*fwl|c12
```

```
twvlc12•ksluv.324
ksluv.o124=•124•1/o324• 1. 134.000/o2•1
twllc12c•ksluv.125•fwllc12c
twvlc12c•ksluv.325
ksluv.o125=•125•1/o325• 1. 134.000/o2•1
twllc12e•ksluv.126•fwllc12e
twvlc12e•ksluv.326
ksluv.o126=•126•1/o326• 1. 134.000/o2•1
twllc12f•ksluv.127•fwllc12f
twvlc12f•ksluv.327
ksluv.o127=•127•1/o327• 1. 134.000/o2•1
twllc12g•ksluv.128•fwllc12g
twvlc12g•ksluv.328
ksluv.o128=•128•1/o328• 1. 134.000/o2•1
twllc12h•ksluv.129•fwllc12h
twvlc12h•ksluv.329
ksluv.o129=•129•1/o329• 1. 134.000/o2•1
twllc14•ksluv.130•fwllc14
twvlc14•ksluv.330
ksluv.o130=•130•1/o330• 1. 134.000/o2•1
twllc14e•ksluv.131•fwllc14e
twvlc14e•ksluv.331
ksluv.o131=•131•1/o331• 1. 134.000/o2•1
twllc14b•ksluv.132•fwllc14b
twvlc14b•ksluv.332
ksluv.o132=•132•1/o332• 1. 134.000/o2•1
ksluv.•2•ksluv.o2
ksluv.o2=•2•1
ksluv.•2= •1•1

' prelseg Locks cpseg,pseg 1,pseg 2,seg
prelseg.•1=• o101• .250, 35.000
prelseg.•1= o102• .250, 35.000
prelseg.•1= o103• .250, 35.000
prelseg.•1= o104• .250, 35.000
twllcpseg•prelseg.101•fwllcpseg
twvlcpseg•prelseg.301
prelseg.o101=•101•1/o301• 1. 100.000/o2•1
tlklpseg 1•prelseg.102•flklpseg 1
ulklpseg 1•prelseg.302
prelseg.o102=•102•1/o302• 1. 100.000/o2•1
tlklpseg 2•prelseg.103•flklpseg 2
ulklpseg 2•prelseg.303
prelseg.o103=•103•1/o303• 1. 100.000/o2•1
tlklseg•prelseg.104•flklseg
ulklseg•prelseg.304
prelseg.o104=•104•1/o304• 1. 100.000/o2•1
prelseg.•2•prelseg.o2
prelseg.o2=•2•1
kslseg.1•vblkslseg•prelseg.901
prelseg.•2=o901•1/o3•1
prelseg.•3•prelseg.o3
prelseg.o3=•3•1
prelseg.•3= •1•1

' kslseg Locks seg
kslseg.•1=• o101• 1
tlklseg•kslseg.101•flklseg
ulklseg•kslseg.301
kslseg.o101=•101•1/o301• 1. 400.000/o2•1
kslseg.•2•kslseg.o2
kslseg.o2=•2•1
kslseg.•2= o102• 1
tlklseg•kslseg.102•flklseg
ulklseg•kslseg.302
kslseg.o102=•102•1/o302• 1. 400.000/o3•1
kslseg.•3•kslseg.o3
kslseg.o3=•3•1
kslseg.•3= o103• 1
tlklseg•kslseg.103•flklseg
ulklseg•kslseg.303
kslseg.o103=•103•1/o303• 1. 400.000/o4•1
kslseg.•4•kslseg.o4
kslseg.o4=•4•1
kslseg.•4= o104• 1
tlklseg•kslseg.104•flklseg
ulklseg•kslseg.304
kslseg.o104=•104•1/o304• 1. 400.000/o5•1
kslseg.•5•kslseg.o5
kslseg.o5=•5•1
kslseg.•5= •1•1

' prelutb Locks c4,c5
prelutb.•1=• o101• .500
prelutb.•1= o102• .500
twllc4•prelutb.101•fwllc4
twvlc4•prelutb.301
prelutb.o101=•101•1/o301• 1. 50.000/o2•1
twllc5•prelutb.102•fwllc5
twvlc5•prelutb.302
prelutb.o102=•102•1/o302• 1. 50.000/o2•1
prelutb.•2•prelutb.o2
prelutb.o2=•2•1
kslutb.1•vblkslutb•prelutb.901
prelutb.•2=o901• .057.60/o3•1
prelutb.•2=o3• .943.60
prelutb.•3•prelutb.o3
prelutb.o3=•3•1
prelutb.•3= •1•1

' kslutb Locks c2,c3,c3e,c3b,c4,c5,c6,
'      c6e,c6f,c6g,c6h,c13,c13e,c13b
kslutb.•1=• o101• .071
kslutb.•1= o102• .071
kslutb.•1= o103• .071
kslutb.•1= o104• .071
kslutb.•1= o105• .071
kslutb.•1= o106• .071
kslutb.•1= o107• .071
kslutb.•1= o108• .071
kslutb.•1= o109• .072
kslutb.•1= o110• .072
kslutb.•1= o111• .072
kslutb.•1= o112• .072
kslutb.•1= o113• .072
kslutb.•1= o114• .072
twllc2•kslutb.101•fwllc2
twvlc2•kslutb.301
kslutb.o101=•101•1/o301• 1. 30.000/o2•1
twllc3•kslutb.102•fwllc3
twvlc3•kslutb.302
kslutb.o102=•102•1/o302• 1. 30.000/o2•1
twllc3e•kslutb.103•fwllc3e
twvlc3e•kslutb.303
kslutb.o103=•103•1/o303• 1. 30.000/o2•1
twllc3b•kslutb.104•fwllc3b
twvlc3b•kslutb.304
kslutb.o104=•104•1/o304• 1. 30.000/o2•1
twllc4•kslutb.105•fwllc4
twvlc4•kslutb.305
kslutb.o105=•105•1/o305• 1. 30.000/o2•1
twllc5•kslutb.106•fwllc5
twvlc5•kslutb.306
kslutb.o106=•106•1/o306• 1. 30.000/o2•1
twllc6•kslutb.107•fwllc6
twvlc6•kslutb.307
kslutb.o107=•107•1/o307• 1. 30.000/o2•1
twllc6e•kslutb.108•fwllc6e
twvlc6e•kslutb.308
kslutb.o108=•108•1/o308• 1. 30.000/o2•1
twllc6f•kslutb.109•fwllc6f
twvlc6f•kslutb.309
kslutb.o109=•109•1/o309• 1. 30.000/o2•1
twllc6g•kslutb.110•fwllc6g
twvlc6g•kslutb.310
kslutb.o110=•110•1/o310• 1. 30.000/o2•1
twllc6h•kslutb.111•fwllc6h
twvlc6h•kslutb.311
kslutb.o111=•111•1/o311• 1. 30.000/o2•1
twllc13•kslutb.112•fwllc13
twvlc13•kslutb.312
kslutb.o112=•112•1/o312• 1. 30.000/o2•1
twllc13e•kslutb.113•fwllc13e
twvlc13e•kslutb.313
kslutb.o113=•113•1/o313• 1. 30.000/o2•1
twllc13b•kslutb.114•fwllc13b
twvlc13b•kslutb.314
kslutb.o114=•114•1/o314• 1. 30.000/o2•1
kslutb.•2•kslutb.o2
kslutb.o2=•2•1
kslutb.•2= o115• .071, 30.000
kslutb.•2= o116• .071, 30.000
kslutb.•2= o117• .071, 30.000
kslutb.•2= o118• .071, 30.000
```

```
ks|utb.i2= o119: .071, 30.000
ks|utb.i2= o120: .071, 30.000
ks|utb.i2= o121: .071, 30.000
ks|utb.i2= o122: .071, 30.000
ks|utb.i2= o123: .072, 30.000
ks|utb.i2= o124: .072, 30.000
ks|utb.i2= o125: .072, 30.000
ks|utb.i2= o126: .072, 30.000
ks|utb.i2= o127: .072, 30.000
ks|utb.i2= o128: .072, 30.000
fw|lc2-ks|utb.115-fw|lc2
tw|lc2-ks|utb.315
ks|utb.o115=:115:1/o315: 1, 150.000/o3:1
fw|lc3-ks|utb.116-fw|lc3
tw|lc3-ks|utb.316
ks|utb.o116=:116:1/o316: 1, 150.000/o3:1
fw|lc3a-ks|utb.117-fw|lc3a
tw|lc3a-ks|utb.317
ks|utb.o117=:117:1/o317: 1, 150.000/o3:1
fw|lc3b-ks|utb.118-fw|lc3b
tw|lc3b-ks|utb.318
ks|utb.o118=:118:1/o318: 1, 150.000/o3:1
fw|lc4-ks|utb.119-fw|lc4
tw|lc4-ks|utb.319
ks|utb.o119=:119:1/o319: 1, 150.000/o3:1
fw|lc5-ks|utb.120-fw|lc5
tw|lc5-ks|utb.320
ks|utb.o120=:120:1/o320: 1, 150.000/o3:1
fw|lc6-ks|utb.121-fw|lc6
tw|lc6-ks|utb.321
ks|utb.o121=:121:1/o321: 1, 150.000/o3:1
fw|lc6a-ks|utb.122-fw|lc6a
tw|lc6a-ks|utb.322
ks|utb.o122=:122:1/o322: 1, 150.000/o3:1
fw|lc6f-ks|utb.123-fw|lc6f
tw|lc6f-ks|utb.323
ks|utb.o123=:123:1/o323: 1, 150.000/o3:1
fw|lc6g-ks|utb.124-fw|lc6g
tw|lc6g-ks|utb.324
ks|utb.o124=:124:1/o324: 1, 150.000/o3:1
fw|lc6h-ks|utb.125-fw|lc6h
tw|lc6h-ks|utb.325
ks|utb.o125=:125:1/o325: 1, 150.000/o3:1
fw|lc13-ks|utb.126-fw|lc13
tw|lc13-ks|utb.326
ks|utb.o126=:126:1/o326: 1, 150.000/o3:1
fw|lc13a-ks|utb.127-fw|lc13a
tw|lc13a-ks|utb.327
ks|utb.o127=:127:1/o327: 1, 150.000/o3:1
fw|lc13b-ks|utb.128-fw|lc13b
tw|lc13b-ks|utb.328
ks|utb.o128=:128:1/o328: 1, 150.000/o3:1
ks|utb.i3-ks|utb.o3
ks|utb.o3=:3:1
ks|utb.i3= o129: .071, 30.000
ks|utb.i3= o130: .071, 30.000
ks|utb.i3= o131: .071, 30.000
ks|utb.i3= o132: .071, 30.000
ks|utb.i3= o133: .071, 30.000
ks|utb.i3= o134: .071, 30.000
ks|utb.i3= o135: .071, 30.000
ks|utb.i3= o136: .071, 30.000
ks|utb.i3= o137: .072, 30.000
ks|utb.i3= o138: .072, 30.000
ks|utb.i3= o139: .072, 30.000
ks|utb.i3= o140: .072, 30.000
ks|utb.i3= o141: .072, 30.000
ks|utb.i3= o142: .072, 30.000
tw|lc2-ks|utb.129-fw|lc2
tw|lc2-ks|utb.329
ks|utb.o129=:129:1/o329: 1, 150.000/o4:1
fw|lc3-ks|utb.130-fw|lc3
tw|lc3-ks|utb.330
ks|utb.o130=:130:1/o330: 1, 150.000/o4:1
fw|lc3a-ks|utb.131-fw|lc3a
tw|lc3a-ks|utb.331
ks|utb.o131=:131:1/o331: 1, 150.000/o4:1
fw|lc3b-ks|utb.132-fw|lc3b
tw|lc3b-ks|utb.332
ks|utb.o132=:132:1/o332: 1, 150.000/o4:1
fw|lc4-ks|utb.133-fw|lc4
tw|lc4-ks|utb.333
ks|utb.o133=:133:1/o333: 1, 150.000/o4:1
fw|lc5-ks|utb.134-fw|lc5
tw|lc5-ks|utb.334
ks|utb.o134=:134:1/o334: 1, 150.000/o4:1
fw|lc6-ks|utb.135-fw|lc6
tw|lc6-ks|utb.335
ks|utb.o135=:135:1/o335: 1, 150.000/o4:1
fw|lc6e-ks|utb.136-fw|lc6e
tw|lc6e-ks|utb.336
ks|utb.o136=:136:1/o336: 1, 150.000/o4:1
fw|lc6f-ks|utb.137-fw|lc6f
tw|lc6f-ks|utb.337
ks|utb.o137=:137:1/o337: 1, 150.000/o4:1
fw|lc6g-ks|utb.138-fw|lc6g
tw|lc6g-ks|utb.338
ks|utb.o138=:138:1/o338: 1, 150.000/o4:1
fw|lc6h-ks|utb.139-fw|lc6h
tw|lc6h-ks|utb.339
ks|utb.o139=:139:1/o339: 1, 150.000/o4:1
fw|lc13-ks|utb.140-fw|lc13
tw|lc13-ks|utb.340
ks|utb.o140=:140:1/o340: 1, 150.000/o4:1
fw|lc13e-ks|utb.141-fw|lc13e
tw|lc13e-ks|utb.341
ks|utb.o141=:141:1/o341: 1, 150.000/o4:1
fw|lc13b-ks|utb.142-fw|lc13b
tw|lc13b-ks|utb.342
ks|utb.o142=:142:1/o342: 1, 150.000/o4:1
ks|utb.i4-ks|utb.o4
ks|utb.o4=:4:1
ks|utb.i4= :1:1

! prelpsc Locks c3,c3e,c3b,c7,c7e,c7b
prelpsc.i1=* o101: .166
prelpsc.i1= o102: .166
prelpsc.i1= o103: .167
prelpsc.i1= o104: .167
prelpsc.i1= o105: .167
prelpsc.i1= o106: .167
fw|lc3-prelpsc.101-fw|lc3
tw|lc3-prelpsc.301
prelpsc.o101=:101:1/o301: 1, 50.000/o2:1
fw|lc3e-prelpsc.102-fw|lc3e
tw|lc3e-prelpsc.302
prelpsc.o102=:102:1/o302: 1, 50.000/o2:1
fw|lc3b-prelpsc.103-fw|lc3b
tw|lc3b-prelpsc.303
prelpsc.o103=:103:1/o303: 1, 50.000/o2:1
fw|lc7-prelpsc.104-fw|lc7
tw|lc7-prelpsc.304
prelpsc.o104=:104:1/o304: 1, 50.000/o2:1
fw|lc7e-prelpsc.105-fw|lc7e
tw|lc7e-prelpsc.305
prelpsc.o105=:105:1/o305: 1, 50.000/o2:1
fw|lc7b-prelpsc.106-fw|lc7b
tw|lc7b-prelpsc.306
prelpsc.o106=:106:1/o306: 1, 50.000/o2:1
prelpsc.i2-prelpsc.o2
prelpsc.o2=:2:1
ks|search.i-vk|ks|search-prelpsc.901
prelpsc.i2=o901: .130.135/o3:1
ks|time.i-vk|ks|time-prelpsc.902
prelpsc.i2=o902: .130.135/o3:1
prelpsc.i2=o3: .740.135
prelpsc.i3-prelpsc.o3
prelpsc.o3=:3:1
prelpsc.i3= :1:1

! ks|search Locks c2,c3,c3e,c3b,c4,c5,
!       c6,c6e,c6f,c6g,c6h,c7,c7e,c7b,
!       c8,c13,c13e,c13b
ks|search.i1=* o101: .055, 50.000
ks|search.i1= o102: .055, 50.000
ks|search.i1= o103: .055, 50.000
ks|search.i1= o104: .055, 50.000
ks|search.i1= o105: .055, 50.000
ks|search.i1= o106: .055, 50.000
ks|search.i1= o107: .055, 50.000
ks|search.i1= o108: .055, 50.000
ks|search.i1= o109: .056, 60.000
ks|search.i1= o110: .056, 50.000
```

```
ks|search.i1= o111: .056, $0.000
ks|search.i1= o112: .056, $0.000
ks|search.i1= o113: .056, $0.000
ks|search.i1= o114: .056, $0.000
ks|search.i1= o115: .056, $0.000
ks|search.i1= o116: .056, $0.000
ks|search.i1= o117: .056, $0.000
ks|search.i1= o118: .056, $0.000
twl|c2>ks|search.101>fwl|c2
twu|c2>ks|search.301
ks|search.o101=>101:1/o301: 1, 1100.000/o2:1
twl|c3>ks|search.102>fwl|c3
twu|c3>ks|search.302
ks|search.o102=>102:1/o302: 1, 1100.000/o2:1
twl|c3e>ks|search.103>fwl|c3e
twu|c3e>ks|search.303
ks|search.o103=>103:1/o303: 1, 1100.000/o2:1
twl|c3b>ks|search.104>fwl|c3b
twu|c3b>ks|search.304
ks|search.o104=>104:1/o304: 1, 1100.000/o2:1
twl|c4>ks|search.105>fwl|c4
twu|c4>ks|search.305
ks|search.o105=>105:1/o305: 1, 1100.000/o2:1
twl|c5>ks|search.106>fwl|c5
twu|c5>ks|search.306
ks|search.o106=>106:1/o306: 1, 1100.000/o2:1
twl|c6>ks|search.107>fwl|c6
twu|c6>ks|search.307
ks|search.o107=>107:1/o307: 1, 1100.000/o2:1
twl|c6e>ks|search.108>fwl|c6e
twu|c6e>ks|search.308
ks|search.o108=>108:1/o308: 1, 1100.000/o2:1
twl|c6f>ks|search.109>fwl|c6f
twu|c6f>ks|search.309
ks|search.o109=>109:1/o309: 1, 1100.000/o2:1
twl|c6g>ks|search.110>fwl|c6g
twu|c6g>ks|search.310
ks|search.o110=>110:1/o310: 1, 1100.000/o2:1
twl|c6h>ks|search.111>fwl|c6h
twu|c6h>ks|search.311
ks|search.o111=>111:1/o311: 1, 1100.000/o2:1
twl|c7>ks|search.112>fwl|c7
twu|c7>ks|search.312
ks|search.o112=>112:1/o312: 1, 1100.000/o2:1
twl|c7e>ks|search.113>fwl|c7e
twu|c7e>ks|search.313
ks|search.o113=>113:1/o313: 1, 1100.000/o2:1
twl|c7b>ks|search.114>fwl|c7b
twu|c7b>ks|search.314
ks|search.o114=>114:1/o314: 1, 1100.000/o2:1
twl|c8>ks|search.115>fwl|c8
twu|c8>ks|search.315
ks|search.o115=>115:1/o315: 1, 1100.000/o2:1
twl|c13>ks|search.116>fwl|c13
twu|c13>ks|search.316
ks|search.o116=>116:1/o316: 1, 1100.000/o2:1
twl|c13e>ks|search.117>fwl|c13e
twu|c13e>ks|search.317
ks|search.o117=>117:1/o317: 1, 1100.000/o2:1
twl|c13b>ks|search.118>fwl|c13b
twu|c13b>ks|search.318
ks|search.o118=>118:1/o318: 1, 1100.000/o2:1
ks|search.i2>ks|search.o2
ks|search.o2=>2:1
ks|search.i2= o119: .055, 200.000
ks|search.i2= o120: .055, 200.000
ks|search.i2= o121: .055, 200.000
ks|search.i2= o122: .055, 200.000
ks|search.i2= o123: .055, 200.000
ks|search.i2= o124: .055, 200.000
ks|search.i2= o125: .055, 200.000
ks|search.i2= o126: .055, 200.000
ks|search.i2= o127: .056, 200.000
ks|search.i2= o128: .056, 200.000
ks|search.i2= o129: .056, 200.000
ks|search.i2= o130: .056, 200.000
ks|search.i2= o131: .056, 200.000
ks|search.i2= o132: .056, 200.000
ks|search.i2= o133: .056, 200.000
ks|search.i2= o134: .056, 200.000
ks|search.i2= o135: .056, 200.000
ks|search.i2= o136: .056, 200.000
twl|c2>ks|search.119>fwl|c2
twu|c2>ks|search.319
ks|search.o119=>119:1/o319: 1, 1100.000/o3:1
twl|c3>ks|search.120>fwl|c3
twu|c3>ks|search.320
ks|search.o120=>120:1/o320: 1, 1100.000/o3:1
twl|c3e>ks|search.121>fwl|c3e
twu|c3e>ks|search.321
ks|search.o121=>121:1/o321: 1, 1100.000/o3:1
twl|c3b>ks|search.122>fwl|c3b
twu|c3b>ks|search.322
ks|search.o122=>122:1/o322: 1, 1100.000/o3:1
twl|c4>ks|search.123>fwl|c4
twu|c4>ks|search.323
ks|search.o123=>123:1/o323: 1, 1100.000/o3:1
twl|c5>ks|search.124>fwl|c5
twu|c5>ks|search.324
ks|search.o124=>124:1/o324: 1, 1100.000/o3:1
twl|c6>ks|search.125>fwl|c6
twu|c6>ks|search.325
ks|search.o125=>125:1/o325: 1, 1100.000/o3:1
twl|c6e>ks|search.126>fwl|c6e
twu|c6e>ks|search.326
ks|search.o126=>126:1/o326: 1, 1100.000/o3:1
twl|c6f>ks|search.127>fwl|c6f
twu|c6f>ks|search.327
ks|search.o127=>127:1/o327: 1, 1100.000/o3:1
twl|c6g>ks|search.128>fwl|c6g
twu|c6g>ks|search.328
ks|search.o128=>128:1/o328: 1, 1100.000/o3:1
twl|c6h>ks|search.129>fwl|c6h
twu|c6h>ks|search.329
ks|search.o129=>129:1/o329: 1, 1100.000/o3:1
twl|c7>ks|search.130>fwl|c7
twu|c7>ks|search.330
ks|search.o130=>130:1/o330: 1, 1100.000/o3:1
twl|c7e>ks|search.131>fwl|c7e
twu|c7e>ks|search.331
ks|search.o131=>131:1/o331: 1, 1100.000/o3:1
twl|c7b>ks|search.132>fwl|c7b
twu|c7b>ks|search.332
ks|search.o132=>132 1/o332: 1, 1100.000/o3:1
twl|c8>ks|search.133>fwl|c8
twu|c8>ks|search.333
ks|search.o133=>133:1/o333: 1, 1100.000/o3:1
twl|c13>ks|search.134>fwl|c13
twu|c13>ks|search.334
ks|search.o134=>134:1/o334: 1, 1100.000/o3:1
twl|c13e>ks|search.135>fwl|c13e
twu|c13e>ks|search.335
ks|search.o135=>135:1/o335: 1, 1100.000/o3:1
twl|c13b>ks|search.136>fwl|c13b
twu|c13b>ks|search.336
ks|search.o136=>136:1/o336: 1, 1100.000/o3:1
ks|search.i3>ks|search.o3
ks|search.o3=>3:1
ks|search.i3= :1:1

! ks|time Locks c2,c3,c3e,c3b,c4,c5,
!       c6,c6e,c6f,c6g,c6h,c7,c7e,
!       c7b,c8,c13,c13e,c13b
ks|time.i1= o101: .055, $0.000
ks|time.i1= o102: .055, $0.000
ks|time.i1= o103: .055, $0.000
ks|time.i1= o104: .055, $0.000
ks|time.i1= o105: .055, $0.000
ks|time.i1= o106: .055, $0.000
ks|time.i1= o107: .055, $0.000
ks|time.i1= o108: .055, $0.000
ks|time.i1= o109: .056, $0.000
ks|time.i1= o110: .056, $0.000
ks|time.i1= o111: .056, $0.000
ks|time.i1= o112: .056, $0.000
ks|time.i1= o113: .056, $0.000
ks|time.i1= o114: .056, $0.000
ks|time.i1= o115: .056, $0.000
ks|time.i1= o116: .056, $0.000
ks|time.i1= o117: .056, $0.000
ks|time.i1= o118: .056, $0.000
twl|c2>ks|time.101>fwl|c2
twu|c2>ks|time.301
ks|time.o101=>101:1/o301: 1, 225.000/o2:1
```

```
twllc3-ksltime.102-fwllc3
twulc3-ksltime.302
ksltime.o102-(102:1/o302: 1. 225.000/o2:1
twllc3a-ksltime.103-fwllc3a
twulc3a-ksltime.303
ksltime.o103-(103:1/o303: 1. 225.000/o2:1
twllc3b-ksltime.104-fwllc3b
twulc3b-ksltime.304
ksltime.o104-(104:1/o304: 1. 225.000/o2:1
twllc4-ksltime.105-fwllc4
twulc4-ksltime.305
ksltime.o105-(105:1/o305: 1. 225.000/o2:1
twllc5-ksltime.106-fwllc5
twulc5-ksltime.306
ksltime.o106-(106:1/o306: 1. 225.000/o2:1
twllc6-ksltime.107-fwllc6
twulc6-ksltime.307
ksltime.o107-(107:1/o307: 1. 225.000/o2:1
twllc6a-ksltime.108-fwllc6a
twulc6a-ksltime.308
ksltime.o108-(108:1/o308: 1. 225.000/o2:1
twllc6f-ksltime.109-fwllc6f
twulc6f-ksltime.309
ksltime.o109-(109:1/o309: 1. 225.000/o2:1
twllc6g-ksltime.110-fwllc6g
twulc6g-ksltime.310
ksltime.o110-(110:1/o310: 1. 225.000/o2:1
twllc6h-ksltime.111-fwllc6h
twulc6h-ksltime.311
ksltime.o111-(111:1/o311: 1. 225.000/o2:1
twllc7-ksltime.112-fwllc7
twulc7-ksltime.312
ksltime.o112-(112:1/o312: 1. 225.000/o2:1
twllc7a-ksltime.113-fwllc7a
twulc7a-ksltime.313
ksltime.o113-(113:1/o313: 1. 225.000/o2:1
twllc7b-ksltime.114-fwllc7b
twulc7b-ksltime.314
ksltime.o114-(114:1/o314: 1. 225.000/o2:1
twllc8-ksltime.115-fwllc8
twulc8-ksltime.315
ksltime.o115-(115:1/o315: 1. 225.000/o2:1
twllc13-ksltime.116-fwllc13
twulc13-ksltime.316
ksltime.o116-(116:1/o316: 1. 225.000/o2:1
twllc13a-ksltime.117-fwllc13a
twulc13a-ksltime.317
ksltime.o117-(117:1/o317: 1. 225.000/o2:1
twllc13b-ksltime.118-fwllc13b
twulc13b-ksltime.318
ksltime.o118-(118:1/o318: 1. 225.000/o2:1
ksltime.(2-ksltime.o2
ksltime.o2-(2:1
ksltime.(2- o119: .055. 75.000
ksltime.(2- o120: .055. 75.000
ksltime.(2- o121: .055. 75.000
ksltime.(2- o122: .055. 75.000
ksltime.(2- o123: .055. 75.000
ksltime.(2- o124: .055. 75.000
ksltime.(2- o125: .055. 75.000
ksltime.(2- o126: .055. 75.000
ksltime.(2- o127: .056. 75.000
ksltime.(2- o128: .056. 75.000
ksltime.(2- o129: .056. 75.000
ksltime.(2- o130: .056. 75.000
ksltime.(2- o131: .056. 75.000
ksltime.(2- o132: .056. 75.000
ksltime.(2- o133: .056. 75.000
ksltime.(2- o134: .056. 75.000
ksltime.(2- o135: .056. 75.000
ksltime.(2- o136: .056. 75.000
twllc2-ksltime.119-fwllc2
twulc2-ksltime.319
ksltime.o119-(119:1/o319: 1. 225.000/o3:1
twllc3-ksltime.120-fwllc3
twulc3-ksltime.320
ksltime.o120-(120:1/o320: 1. 225.000/o3:1
twllc3a-ksltime.121-fwllc3a
twulc3a-ksltime.321
ksltime.o121-(121:1/o321: 1. 225.000/o3:1
twllc3b-ksltime.122-fwllc3b
twulc3b-ksltime.322
```

```
bs|time.o122=:122:1/o322: 1. 225.000/o3:1
twu|c4=bs|time.123=fwl|c4
twu|c4=bs|time.323
bs|time.o123=:123:1/o323: 1. 225.000/o3:1
twu|c5=bs|time.124=fwl|c5
twu|c5=bs|time.324
bs|time.o124=:124:1/o324: 1. 225.000/o3:1
twl|c6=bs|time.125=fwl|c6
twu|c6=bs|time.325
bs|time.o125=:125:1/o325: 1. 225.000/o3:1
twl|c6e=bs|time.126=fwl|c6e
twu|c6e=bs|time.326
bs|time.o126=:126:1/o326: 1. 225.000/o3:1
twl|c6f=bs|time.127=fwl|c6f
twu|c6f=bs|time.327
bs|time.o127=:127:1/o327: 1. 225.000/o3:1
twl|c6g=bs|time.128=fwl|c6g
twu|c6g=bs|time.328
bs|time.o128=:128:1/o328: 1. 225.000/o3:1
twl|c6h=bs|time.129=fwl|c6h
twu|c6h=bs|time.329
bs|time.o129=:129:1/o329: 1. 225.000/o3:1
twl|c7=bs|time.130=fwl|c7
twu|c7=bs|time.330
bs|time.o130=:130:1/o330: 1. 225.000/o3:1
twl|c7e=bs|time.131=fwl|c7e
twu|c7e=bs|time.331
bs|time.o131=:131:1/o331: 1. 225.000/o3:1
twl|c7b=bs|time.132=fwl|c7b
twu|c7b=bs|time.332
bs|time.o132=:132:1/o332: 1. 225.000/o3:1
twl|c8=bs|time.133=fwl|c8
twu|c8=bs|time.333
bs|time.o133=:133:1/o333: 1. 225.000/o3:1
twl|c13=bs|time.134=fwl|c13
twu|c13=bs|time.334
bs|time.o134=:134:1/o334: 1. 225.000/o3:1
twl|c13e=bs|time.135=fwl|c13e
twu|c13e=bs|time.335
bs|time.o135=:135:1/o335: 1. 225.000/o3:1
twl|c13b=bs|time.136=fwl|c13b
twu|c13b=bs|time.336
bs|time.o136=:136:1/o336: 1. 225.000/o3:1
bs|time.:3=bs|time.o3
bs|time.o3=:3:1
bs|time.:3=:1:1

! Lexicon lock lk|seg
flk|seg=lk|seg.1=tlk|seg
lk|seg.2=ulk|seg
lk|seg.:1=*:2:1 / o:1:1 / :1:1
Att ulk|seg queue:100, vol:1
Sched noncom lk|seg
Collect nostat lk|seg

! Lexicon lock lk|pseg 1
flk|pseg 1=lk|pseg 1.1=tlk|pseg 1
lk|pseg 1.2=ulk|pseg 1
lk|pseg 1.:1=*:2:1 / o:1:1 / :1:1
Att ulk|pseg 1 queue:100, vol:1
Sched noncom lk|pseg 1
Collect nostat lk|pseg 1

! Lexicon lock lk|pseg 2
flk|pseg 2=lk|pseg 2.1=tlk|pseg 2
lk|pseg 2.2=ulk|pseg 2
lk|pseg 2.:1=*:2:1 / o:1:1 / :1:1
Att ulk|pseg 2 queue:100, vol:1
Sched noncom lk|pseg 2
Collect nostat lk|pseg 2

! Lexicon lock lk|mxn 1
flk|mxn 1=lk|mxn 1.1=tlk|mxn 1
lk|mxn 1.2=ulk|mxn 1
lk|mxn 1.:1=*:2:1 / o:1:1 / :1:1
Att ulk|mxn 1 queue:100, vol:1
Sched noncom lk|mxn 1
Collect nostat lk|mxn 1

! Lexicon lock lk|mxn 2
flk|mxn 2=lk|mxn 2.1=tlk|mxn 2
lk|mxn 2.2=ulk|mxn 2
lk|mxn 2.:1=*:2:1 / o:1:1 / :1:1
Att ulk|mxn 2 queue:100, vol:1
Sched noncom lk|mxn 2
Collect nostat lk|mxn 2

! Lexicon lock lk|phon
flk|phon=lk|phon.1=tlk|phon
lk|phon.2=ulk|phon
lk|phon.:1=*:2:1 / o:1:1 / :1:1
Att ulk|phon queue:100, vol:1
Sched noncom lk|phon
Collect nostat lk|phon

! Lexicon lock lk|surn
flk|surn=lk|surn.1=tlk|surn
lk|surn.2=ulk|surn
lk|surn.:1=*:2:1 / o:1:1 / :1:1
Att ulk|surn queue:100, vol:1
Sched noncom lk|surn
Collect nostat lk|surn

! Lexicon lock lk|wrdsurn
flk|wrdsurn=lk|wrdsurn.1=tlk|wrdsurn
lk|wrdsurn.2=ulk|wrdsurn
lk|wrdsurn.:1=*:2:1 / o:1:1 / :1:1
Att ulk|wrdsurn queue:100, vol:1
Sched noncom lk|wrdsurn
Collect nostat lk|wrdsurn

! Lexicon lock lk|word
flk|word=lk|word.1=tlk|word
lk|word.2=ulk|word
lk|word.:1=*:2:1 / o:1:1 / :1:1
Att ulk|word queue:100, vol:1
Sched noncom lk|word
Collect nostat lk|word

! Lexicon lock lk|shdword
flk|shdword=lk|shdword.1=tlk|shdword
lk|shdword.2=ulk|shdword
lk|shdword.:1=*:2:1 / o:1:1 / :1:1
Att ulk|shdword queue:100, vol:1
Sched noncom lk|shdword
Collect nostat lk|shdword

! Lexicon lock lk|shdsent
flk|shdsent=lk|shdsent.1=tlk|shdsent
lk|shdsent.2=ulk|shdsent
lk|shdsent.:1=*:2:1 / o:1:1 / :1:1
Att ulk|shdsent queue:100, vol:1
Sched noncom lk|shdsent
Collect nostat lk|shdsent

! C1  word=wrdsurn
full|c1=wlc1.1=twl|c1
wuc1.1=twu|c1
tlk|wrdsurn=wlc1.2=flk|wrdsurn
ulk|wrdsurn=wuc1.2
tlk|word=wlc1.3=flk|word
ulk|word=wuc1.3
wlc1.:1=*o2:1/ :2:1/ o3:1/ :3:1/ o:1:1/ :1:1
wuc1.:1=*o2:1/ o3:1/ :1:1

Attribute vb|lks|elo Queue: 250, Volume:   0, Delay:   .000, Startup:   .000
Attribute vb|lks|csep Queue: 250, Volume:   0, Delay:   .000, Startup:   .000
Attribute vb|lks|psyn Queue: 250, Volume:   0, Delay:   .000, Startup:   .000
Attribute vb|lks|SEAPC Queue: 250, Volume:   0, Delay:   .000, Startup:   .000
Attribute vb|lks|seg Queue: 250, Volume:   0, Delay:   .000, Startup:   .000
Attribute vb|lk|beltime Queue: 250, Volume:   0, Delay:   .000, Startup:   .000
Attribute vb|lks|utb Queue: 250, Volume:   0, Delay:   .000, Startup:   .000
Attribute vb|lks|UV Queue: 250, Volume:   0, Delay:   .000, Startup:   .000
Attribute TOPPE|RLO Queue: 250, Volume:   0, Delay:   .000, Startup:   .000
Attribute TOPPE|PSC Queue: 250, Volume:   0, Delay:   .000, Startup:   .000
Attribute TOPPE|PSYN Queue: 250, Volume:   0, Delay:   .000, Startup:   .000
```

```
Attribute TOPPE|PPOL Queue: 500, Volume:   0, Delay:   .000, Startup:   .000
Attribute TOPPE|SEG Queue: 250, Volume:   0, Delay:   .000, Startup:   .000
Attribute TOPPE|TUTB Queue: 250, Volume:   0, Delay:   .000, Startup:   .000
copy wlcpseg,wlcmxn,wlc2,wlc3,wlc4,wlc5,wlc6,wlc7,wlc8,wlc9,wlc10 : wlc1
copy wucpseg,wucmxn,wuc2,wuc3,wuc4,wuc5,wuc6,wuc7,wuc8,wuc9,wuc10 : wuc1
copy wlc11,wlc12,wlc13,wlc14,wlc14e,wlc14b,wlc13e,wlc13b,wlc12e : wlc1
copy wuc11,wuc12,wuc13,wuc14,wuc14e,wuc14b,wuc13e,wuc13b,wuc12e : wuc1
copy wlc12f,wlc12g,wlc12h,wlc11e,wlc11b,wlc11e,wlc11f,wlc11g : wlc1
copy wuc12f,wuc12g,wuc12h,wuc11e,wuc11b,wuc11e,wuc11f,wuc11g : wuc1
copy wlc11h,wlc10e,wlc10b,wlc9e,wlc9f,wlc9g,wlc9h,wlc7e,wlc7b : wlc1
copy wuc11h,wuc10e,wuc10b,wuc9e,wuc9f,wuc9g,wuc9h,wuc7e,wuc7b : wuc1
copy wlc6e,wlc6f,wlc6g,wlc6h,wlc3e,wlc3b : wlc1
copy wuc6e,wuc6f,wuc6g,wuc6h,wuc3e,wuc3b : wuc1
sched noncompete wlc1,wuc1
coll nostat wlc1,wuc1


' cpseg pseg 2 + pseg 1
fwl|cpseg-wlcpseg.1+twl|cpseg
wucpseg.1+twu|cpseg
t1k|pseg 1+wlcpseg.2+f1k|pseg 1
u1k|pseg 1+wucpseg.2
t1k|pseg 2+wlcpseg.3+f1k|pseg 2
u1k|pseg 2+wucpseg.3
sched noncompete wlcpseg,wucpseg
collect nostat wlcpseg,wucpseg

' cmxn mxn 2 + mxn 1
fwl|cmxn-wlcmxn.1+twl|cmxn
wucmxn.1+twu|cmxn
t1k|mxn 1+wlcmxn.2+f1k|mxn 1
u1k|mxn 1+wucmxn.2
t1k|mxn 2+wlcmxn.3+f1k|mxn 2
u1k|mxn 2+wucmxn.3
sched noncompete wlcmxn,wucmxn
collect nostat wlcmxn,wucmxn

' c2 c1 + surn
fwl|c2-wlc2.1+twl|c2
wuc2.1+twu|c2
t1k|surn-wlc2.2+f1k|surn
u1k|surn-wuc2.2
twl|c1-wlc2.3+fwl|c1
twu|c1-wuc2.3
sched noncompete wlc2,wuc2
collect nostat wlc2,wuc2

' c3 word + c7
fwl|c3-wlc3.1+twl|c3
wuc3.1+twu|c3
twl|c7-wlc3.2+fwl|c7
twu|c7-wuc3.2
t1k|word-wlc3.3+f1k|word
u1k|word-wuc3.3
sched noncompete wlc3,wuc3
collect nostat wlc3,wuc3

' c4 wrdsurn + c8
fwl|c4-wlc4.1+twl|c4
wuc4.1+twu|c4
twl.c8-wlc4.2+fwl|c8
twu|c8-wuc4.2
t1k|wrdsurn-wlc4.3+f1k|wrdsurn
u1k|wrdsurn-wuc4.3
sched noncompete wlc4,wuc4
collect nostat wlc4,wuc4

' c5 wrdsurn + surn
fwl|c5-wlc5.1+twl|c5
wuc5.1+twu|c5
t1k|surn-wlc5.2+f1k|surn
u1k|surn-wuc5.2
t1k|wrdsurn-wlc5.3+f1k|wrdsurn
u1k|wrdsurn-wuc5.3
sched noncompete wlc5,wuc5
collect nostat wlc5,wuc5

' c6 c8 + c12
fwl|c6-wlc6.1+twl|c6
wuc6.1+twu|c6
twl|c12-wlc6.2+fwl|c12
twu|c12-wuc6.2
twl|c8-wlc6.3+fwl|c8
twu|c8-wuc6.3
sched noncompete wlc6,wuc6
collect nostat wlc6,wuc6

' c7 surn + c10
fwl|c7-wlc7.1+twl|c7
wuc7.1+twu|c7
twl|c10-wlc7.2+fwl|c10
twu|c10-wuc7.2
t1k|surn-wlc7.3+f1k|surn
u1k|surn-wuc7.3
sched noncompete wlc7,wuc7
collect nostat wlc7,wuc7

' c8 surn + phon
fwl|c8-wlc8.1+twl|c8
wuc8.1+twu|c8
t1k|phon-wlc8.2+f1k|phon
u1k|phon-wuc8.2
t1k|surn-wlc8.3+f1k|surn
u1k|surn-wuc8.3
sched noncompete wlc8,wuc8
collect nostat wlc8,wuc8

' c9 phon + c12
fwl|c9-wlc9.1+twl|c9
wuc9.1+twu|c9
twl|c12-wlc9.2+fwl|c12
twu|c12-wuc9.2
t1k|phon-wlc9.3+f1k|phon
u1k|phon-wuc9.3
sched noncompete wlc9,wuc9
collect nostat wlc9,wuc9

' c10 phon + cmxn
fwl|c10-wlc10.1+twl|c10
wuc10.1+twu|c10
twl|cmxn-wlc10.2+fwl|cmxn
twu|cmxn-wuc10.2
t1k|phon-wlc10.3+f1k|phon
u1k|phon-wuc10.3
sched noncompete wlc10,wuc10
collect nostat wlc10,wuc10

' c11 cmxn + c14
fwl|c11-wlc11.1+twl|c11
wuc11.1+twu|c11
twl|c14-wlc11.2+fwl|c14
twu|c14-wuc11.2
twl|cmxn-wlc11.3+fwl|cmxn
twu|cmxn-wuc11.3
sched noncompete wlc11,wuc11
collect nostat wlc11,wuc11

' c12 cmxn + cpseg
fwl|c12-wlc12.1+twl|c12
wuc12.1+twu|c12
twl|cpseg-wlc12.2+fwl|cpseg
twu|cpseg-wuc12.2
twl|cmxn-wlc12.3+fwl|cmxn
twu|cmxn-wuc12.3
sched noncompete wlc12,wuc12
collect nostat wlc12,wuc12

' c13 phon + cpseg
fwl|c13-wlc13.1+twl|c13
wuc13.1+twu|c13
twl|cpseg-wlc13.2+fwl|cpseg
twu|cpseg-wuc13.2
t1k|phon-wlc13.3+f1k|phon
u1k|phon-wuc13.3
sched noncompete wlc13,wuc13
```

```
collect nostat wlc13,wuc13

' c14 cpseg * seg
fwllc14*wlc14,1*twllc14
wuc14,1*twulc14
t1k1seg*wlc14,2*f1k1seg
u1k1seg*wuc14,2
twllcpseg*wlc14,3*fwllcpseg
twulcpseg*wuc14,3
sched noncompete wlc14,wuc14
collect nostat wlc14,wuc14

' c14e pseg 1 * seg
fwllc14e*wlc14e,1*twllc14e
wuc14e,1*twulc14e
t1k1seg*wlc14e,2*f1k1seg
u1k1seg*wuc14e,2
t1k1pseg 1*wlc14e,3*f1k1pseg 1
u1k1pseg 1*wuc14e,3
sched noncompete wlc14e,wuc14e
collect nostat wlc14e,wuc14e

' c14b pseg 2 * seg
fwllc14b*wlc14b,1*twllc14b
wuc14b,1*twulc14b
t1k1seg*wlc14b,2*f1k1seg
u1k1seg*wuc14b,2
t1k1pseg 2*wlc14b,3*f1k1pseg 2
u1k1pseg 2*wuc14b,3
sched noncompete wlc14b,wuc14b
collect nostat wlc14b,wuc14b

' c13e phon * pseg 1
fwllc13e*wlc13e,1*twllc13e
wuc13e,1*twulc13e
t1k1pseg 1*wlc13e,2*f1k1pseg 1
u1k1pseg 1*wuc13e,2
t1k1phon*wlc13e,3*f1k1phon
u1k1phon*wuc13e,3
sched noncompete wlc13e,wuc13e
collect nostat wlc13e,wuc13e

' c13b phon * pseg 2
fwllc13b*wlc13b,1*twllc13b
wuc13b,1*twulc13b
t1k1pseg 2*wlc13b,2*f1k1pseg 2
u1k1pseg 2*wuc13b,2
t1k1phon*wlc13b,3*f1k1phon
u1k1phon*wuc13b,3
sched noncompete wlc13b,wuc13b
collect nostat wlc13b,wuc13b

' c12e mxn 1 * pseg 1
fwllc12e*wlc12e,1*twllc12e
wuc12e,1*twulc12e
t1k1pseg 1*wlc12e,2*f1k1pseg 1
u1k1pseg 1*wuc12e,2
t1k1mxn 1*wlc12e,3*f1k1mxn 1
u1k1mxn 1*wuc12e,3
sched noncompete wlc12e,wuc12e
collect nostat wlc12e,wuc12e

' c12f mxn 1 * pseg 2
fwllc12f*wlc12f,1*twllc12f
wuc12f,1*twulc12f
t1k1pseg 2*wlc12f,2*f1k1pseg 2
u1k1pseg 2*wuc12f,2
t1k1mxn 1*wlc12f,3*f1k1mxn 1
u1k1mxn 1*wuc12f,3
sched noncompete wlc12f,wuc12f
collect nostat wlc12f,wuc12f

' c12g mxn 2 * pseg 1
fwllc12g*wlc12g,1*twllc12g
wuc12g,1*twulc12g
t1k1pseg 1*wlc12g,2*f1k1pseg 1
u1k1pseg 1*wuc12g,2
t1k1mxn 2*wlc12g,3*f1k1mxn 2
u1k1mxn 2*wuc12g,3
sched noncompete wlc12g,wuc12g
collect nostat wlc12g,wuc12g

' c12h mxn 2 * pseg 2
fwllc12h*wlc12h,1*twllc12h
wuc12h,1*twulc12h
t1k1pseg 2*wlc12h,2*f1k1pseg 2
u1k1pseg 2*wuc12h,2
t1k1mxn 2*wlc12h,3*f1k1mxn 2
u1k1mxn 2*wuc12h,3
sched noncompete wlc12h,wuc12h
collect nostat wlc12h,wuc12h

' c11e mxn 1 * c14
fwllc11e*wlc11e,1*twllc11e
wuc11e,1*twulc11e
twllc14*wlc11e,2*fwllc14
twulc14*wuc11e,2
t1k1mxn 1*wlc11e,3*f1k1mxn 1
u1k1mxn 1*wuc11e,3
sched noncompete wlc11e,wuc11e
collect nostat wlc11e,wuc11e

' c11b mxn 2 * c14
fwllc11b*wlc11b,1*twllc11b
wuc11b,1*twulc11b
twllc14*wlc11b,2*fwllc14
twulc14*wuc11b,2
t1k1mxn 2*wlc11b,3*f1k1mxn 2
u1k1mxn 2*wuc11b,3
sched noncompete wlc11b,wuc11b
collect nostat wlc11b,wuc11b

' c11e mxn 1 * c14e
fwllc11e*wlc11e,1*twllc11e
wuc11e,1*twulc11e
twllc14e*wlc11e,2*fwllc14e
twulc14e*wuc11e,2
t1k1mxn 1*wlc11e,3*f1k1mxn 1
u1k1mxn 1*wuc11e,3
sched noncompete wlc11e,wuc11e
collect nostat wlc11e,wuc11e

' c11f mxn 1 * c14b
fwllc11f*wlc11f,1*twllc11f
wuc11f,1*twulc11f
twllc14b*wlc11f,2*fwllc14b
twulc14b*wuc11f,2
t1k1mxn 1*wlc11f,3*f1k1mxn 1
u1k1mxn 1*wuc11f,3
sched noncompete wlc11f,wuc11f
collect nostat wlc11f,wuc11f

' c11g mxn 2 * c14e
fwllc11e*wlc11g,1*twllc11g
wuc11g,1*twulc11g
twllc14e*wlc11g,2*fwllc14e
twulc14e*wuc11g,2
t1k1mxn 2*wlc11g,3*f1k1mxn 2
u1k1mxn 2*wuc11g,3
sched noncompete wlc11g,wuc11g
collect nostat wlc11g,wuc11g

' c11h mxn 2 * c14b
fwllc11b*wlc11h,1*twllc11h
wuc11h,1*twulc11h
twllc14b*wlc11h,2*fwllc14b
twulc14b*wuc11h,2
t1k1mxn 2*wlc11h,3*f1k1mxn 2
u1k1mxn 2*wuc11h,3
sched noncompete wlc11h,wuc11h
collect nostat wlc11h,wuc11h

' c10e phon * mxn 1
fwllc10e*wlc10e,1*twllc10e
```

```
wuc10e.1+twulc10e
tlk|wxn 1+wulc10e.2+flk|wxn 1
ulk|wxn 1+wuc10e.2
tlk|phon+wlc10e.3+flk|phon
ulk|phon+wuc10e.3
sched noncompete wlc10e,wuc10e
collect nostat wlc10e,wuc10e

! c10b phon + wxn 2
fwllc10b+wlc10b.1+twllc10b
wuc10b.1+twulc10b
tlk|wxn 2+wlc10b.2+flk|wxn 2
ulk|wxn 2+wuc10b.2
tlk|phon+wlc10b.3+flk|phon
ulk|phon+wuc10b.3
sched noncompete wlc10b,wuc10b
collect nostat wlc10b,wuc10b

! c9e phon + c12e
fwllc9e+wlc9e.1+twllc9e
wuc9e.1+twulc9e
twllc12e+wlc9e.2+fwllc12e
twulc12e+wuc9e.2
tlk|phon+wlc9e.3+flk|phon
ulk|phon+wuc9e.3
sched noncompete wlc9e,wuc9e
collect nostat wlc9e,wuc9e

! c9f phon + c12f
fwllc9f+wlc9f.1+twllc9f
wuc9f.1+twulc9f
twllc12f+wlc9f.2+fwllc12f
twulc12f+wuc9f.2
tlk|phon+wlc9f.3+flk|phon
ulk|phon+wuc9f.3
sched noncompete wlc9f,wuc9f
collect nostat wlc9f,wuc9f

! c9g phon + c12g
fwllc9g+wlc9g.1+twllc9g
wuc9g.1+twulc9g
twllc12g+wlc9g.2+fwllc12g
twulc12g+wuc9g.2
tlk|phon+wlc9g.3+flk|phon
ulk|phon+wuc9g.3
sched noncompete wlc9g,wuc9g
collect nostat wlc9g,wuc9g

! c9h phon + c12h
fwllc9h+wlc9h.1+twllc9h
wuc9h.1+twulc9h
twllc12h+wlc9h.2+fwllc12h
twulc12h+wuc9h.2
tlk|phon+wlc9h.3+flk|phon
ulk|phon+wuc9h.3
sched noncompete wlc9h,wuc9h
collect nostat wlc9h,wuc9h

! c7e surn + c10e
fwllc7e+wlc7e.1+twllc7e
wuc7e.1+twulc7e
twllc10e+wlc7e.2+fwllc10e
twulc10e+wuc7e.2
tlk|surn+wlc7e.3+flk|surn
ulk|surn+wuc7e.3
sched noncompete wlc7e,wuc7e
collect nostat wlc7e,wuc7e

! c7b surn + c10b
fwllc7b+wlc7b.1+twllc7b
wuc7b.1+twulc7b
twllc10b+wlc7b.2+fwllc10b
twulc10b+wuc7b.2
tlk|surn+wlc7b.3+flk|surn
ulk|surn+wuc7b.3
sched noncompete wlc7b,wuc7b
collect nostat wlc7b,wuc7b

! c6e c8 + c12e
fwllc6e+wlc6e.1+twllc6e
wuc6e.1+twulc6e
twllc12e+wlc6e.2+fwllc12e
twulc12e+wuc6e.2
twllc8+wlc6e.3+fwllc8
twulc8+wuc6e.3
sched noncompete wlc6e,wuc6e
collect nostat wlc6e,wuc6e

! c6f c8 + c12f
fwllc6f+wlc6f.1+twllc6f
wuc6f.1+twulc6f
twllc12f+wlc6f.2+fwllc12f
twulc12f+wuc6f.2
twllc8+wlc6f.3+fwllc8
twulc8+wuc6f.3
sched noncompete wlc6f,wuc6f
collect nostat wlc6f,wuc6f

! c6g c8 + c12g
fwllc6g+wlc6g.1+twllc6g
wuc6g.1+twulc6g
twllc12g+wlc6g.2+fwllc12g
twulc12g+wuc6g.2
twllc8+wlc6g.3+fwllc8
twulc8+wuc6g.3
sched noncompete wlc6g,wuc6g
collect nostat wlc6g,wuc6g

! c6h c8 + c12h
fwllc6h+wlc6h.1+twllc6h
wuc6h.1+twulc6h
twllc12h+wlc6h.2+fwllc12h
twulc12h+wuc6h.2
twllc8+wlc6h.3+fwllc8
twulc8+wuc6h.3
sched noncompete wlc6h,wuc6h
collect nostat wlc6h,wuc6h

! c3e word + c7e
fwllc3e+wlc3e.1+twllc3e
wuc3e.1+twulc3e
twllc7e+wlc3e.2+fwllc7e
twulc7e+wuc3e.2
tlk|word+wlc3e.3+flk|word
ulk|word+wuc3e.3
sched noncompete wlc3e,wuc3e
collect nostat wlc3e,wuc3e

! c3b word + c7b
fwllc3b+wlc3b.1+twllc3b
wuc3b.1+twulc3b
twllc7b+wlc3b.2+fwllc7b
twulc7b+wuc3b.2
tlk|word+wlc3b.3+flk|word
ulk|word+wuc3b.3
sched noncompete wlc3b,wuc3b
collect nostat wlc3b,wuc3b
copy ks|elol:ks|elo
copy ks|cesgl:ks|cseg
copy ks|psyn1,ks|psyn2:ks|psyn
copy ks|searchl:ks|search
copy ks|segl,ks|seg2:ks|seg
copy ks|timel:ks|time
copy ks|utbl:ks|utb
copy ks|uvl,ks|uv2,ks|uv3:ks|uv
```

APPENDIX C

Validation of Simulation Results

Both the Bliss/11 and Hearsay II examples of Chapter III relied on the results of statistics obtained through simulations. When using statistics, one must be prepared for the possibility of error. For simulation experiments, two validation factors are required to verify the significance of the results [Gordon 69]. These are:

1. Elimination of initial bias.
2. Development of a confidence interval.

The method chosen for the elimination of initial bias was the use of trial runs to determine simulation run times. Since many simulations were required for the Bliss/11 experiments, a small number of trial runs were used to estimate the simulation run times for all other runs. It was observed that the initial bias was eliminated after about fifty messages entered the RESULT process of the Bliss/11 model. The number of messages used in the experiments ranged from 475 to 850. In contrast with the Bliss/11 message number measure, the Hearsay II simulations were based on simulated time. It was observed that the Hearsay II results stabilized after about 5000 time units. Consequently, the amounts of time used for the simulation experiments ranged from 10,000 to 100,000 time units. Thus, following one of Gordon's recommendations [Gordon 69], the initial bias was eliminated from the experiments.

The determination of confidence intervals for all the simulations would have required a relatively high overhead in the simulations. The standard techniques require either repetitions of a particular simulation using different random number generator seeds or, alternatively, one very long simulation run that is divided into a

set of batches. Considering the large number of Bliss/11 simulations that were performed, it was felt that the development of a confidence interval for one simulation would be used to represent the entire set of Bliss/11 simulations. Figure C-1 shows the data taken from one long simulation run from the Bliss/11 simulation using six processors and FIFO scheduling.

Stop Time	No. FR Sends	No. LS Sends	LEX Time Computing	Percent Thru	Thru Rate
40	75	76	.238	89.3	3.75
60	65	78	.239	77.7	3.25
80	69	83	.241	83.1	3.45
100	53	56	.355	94.1	2.65
120	66	69	.226	74.5	3.30
140	58	50	.248	71.9	2.90
160	58	62	.249	84.7	3.40
180	79	74	.270	106.6	3.95
200	82	79	.253	103.7	4.10
220	68	85	.226	76.8	3.40
240	64	67	.280	89.6	3.20
260	74	80	.234	86.6	3.70

Figure C-1. Bliss/11 FIFO 6 Processors Evaluation Data

From these data, 90% confidence intervals were computed for the LEX Computing Time, [.245, .264]; Percent Thru, [84.6, 88.5]; and Thru Rate, [3.28, 3.57]. From the data shown in Chapter III, it can be seen that each of the values falls within these respective confidence intervals (i.e., .259, 88.3, and 3.28).

Based on the validation of the numeric significance of this selected simulation, the other Bliss/11 simulations are felt also to be valid. This seems reasonable since the various simulation results did not have any unusual patterns.

The same general techniques were used to run the Hearsay II simulations as were used for the Bliss/11 simulations. As discussed earlier, the initial condition bias was eliminated by running the Hearsay II simulation for a long time. Confidence

intervals were not established for the Hearsay II experiment, because the Hearsay II result was based on an accumulated statistic (i.e., average active processors) and the method used for validating this type of statistic required multiple runs [Gordon 69]. Since each Hearsay II experimental run was relatively expensive (from 20 minutes to 1 hour computer run time), this validation was not felt to be worth spending the required resources. Moreover, the Hearsay II simulation results were correspondences to Fennell's simulation experiments, which were also not validated [Fennell 75a].

Bibliography

Adam, T.J. and Chandy, K.M., "Scheduling Processors in Systems with Multiprocessing Architectures," Tech. Report, Computer Science Dept., University of Texas (1972).

Adams, D.A., "A Model for Parallel Computations," in ***Parallel Processor Systems, Technologies and Applications***, L. C. Hobbs, Ed. (1970), 311-333.

Anderson, J.P. et al., "D825 - A Multiple Computer System for Command and Control," ***Proc. FJCC*** Vol. 22 (1962), 86-96.

Anderson, J.P., "Program Structures for Parallel Processing," ***CACM*** Vol. 8 No. 12 (1965), 786-789.

Baer, J.L. and Estrin, G., "Bounds for Maximum Parallelism in a Bilogic Graph Model of Computations," ***IEEE Trans. Comput.*** C-18 (1969), 1012-1014.

Baer, J.L. and Russel, E.C., "Preparation and Evaluation of Computer Programming for Parallel Processing Systems," in ***Parallel Processor Systems, Technologies and Applications***, L. C. Hobbs, Ed. (1970), 375-415.

Baer, J.L., "A Survey of Some Theoretical Aspects of Multiprocessing," ***Computing Surveys*** Vol. 5 No.1 (March 1973), 31-80.

Balzer, R.M., "Ports -- A Method for Dynamic Interprogram Communication and Job Control," ARPA Report No. 189-1 (1971).

Bell, C.G., Grason, J. and Newell, A., ***Designing Computers and Digital Systems***, Digital Press, Maynard, Mass. (1972).

Berge, C., ***Theory of Graphs and its Applications***, John Wiley and Sons, Inc. (1962).

Boehm, B.W., "The High Cost of Software," in ***Proceedings of a Symposium on the High Cost of Software***, J. Goldberg, Ed., Stanford Research Institute, Menlo Park, CA, (Sept. 1973), 27-40.

Bovet, D.P., "Legality of Multilogic Graphs," Institute di Automatica, University di Roma (1969).

Bredt, T.H. and McCluskey, E.M., "Analysis and Synthesis of Control Mechanisms for Parallel Processes," in ***Parallel Processor Systems, Technologies and Applications***, L. C. Hobbs, Ed. (1970), 287-295.

Brinch Hansen, P., "A Programming Methodology for Operating System Design," *Proc. IFIP 74 Congress* (1974), 394-397.

Browne, J.C., Chandy, K.M., Hogarth, J. and Lee, C. C-A., "The Effect of Throughput of Multiprocessing in a Multiprogramming Environment," *IEEE Trans. Comput.* Vol C-22 No. 8 (Aug. 1973), 728-735.

Clark, W.A. and Molnar, C.E., "The Promise of Macromodular Systems," *IEEE CompCon 72* (Sept. 1972), 309-312.

Conway, M.E., "A Multiprocessor System Design," *AFIPS Proc. FJCC* Vol. 24 (1966), 139-163.

Courtois, P.J., "Instabilities and Saturation in Multiprocessing Computer Systems," Notes for Advanced Course on Computer Systems held in Alpe d'Huiz (Grenoble), Dec. 1972.

Dahl, O.J., Dijkstra, E.W. and Hoare, C.A.R., *Structured Programming*, Academic Press, New York (1972).

Dennis, J.B., "Modular, Asynchronous Control Structures for a High Performance Processor," in *Concurrent Systems and Parallel Computation Conference*, ACM (1970), 55-80.

Dennis, J.B., "Coroutines and Parallel Computations," Fifth Annual Princeton Conf. on Inf. Sciences and Systems, 1971.

Dennis, J.B., "First Version of Data Flow Procedure Language," Project MAC computation structures memo No. 93 (Nov. 1973) (a).

Dennis, J.B. and Fosseen, J.B., "Introduction to Data Flow Schema," Project MAC computation structures memo No. 81-1 (Sept. 1973) (b).

Dijkstra, E.W., "Self-stabilizing Systems in Spite of Distributed Control," *CACM* Vol. 17 No. 11 (Nov. 1974), 643-644.

Erman, L.D., Fennell, R.D., Lessor, V.R. and Reddy, D.R., "System Organizations for Speech Understanding: Implications of Network and Multiprocessor Computer Architectures for AI," *Proc. 3rd IJCAI*, Stanford, Calif. (1973), 194-199.

Erman, L.D. and Lessor, V.R., "A Multi-level Organization for Problem Solving Using Many, Diverse, Cooperating Sources of Knowledge," Tech. Report, Computer Science Dept., Carnegie-Mellon University (1975).

Estrin, G. and Turn, R., "Automatic Assignment of Computations in a Variable Structure Computer System," *IEEE Trans. on Electronic Computers* EC-12 (Dec. 1963), 756-773.

Farber, D.J., "A Ring Network," *DATAMATION* Vol. 21 No. 2 (Feb. 1975), 44-46.

Fennell, R.D., "Multiprocess Software Architecture for AI Problem Solving," Ph. D. Thesis, Computer Science Dept., Carnegie-Mellon University (1975) (a).

Fennell, R.D. and Lesser, V.R., "Parallelism in AI Problem Solving: A Case Study of Hearsay II," Tech. Report, Computer Science Dept., Carnegie-Mellon University (1975) (b).

Fishman, G.S., ***Concepts and Methods in Discrete Event Digital Simulation***, J. Wiley & Sons (1973).

Flynn, M.J., "Very High-Speed Computing Systems," ***Proc. IEEE*** Vol. 54 No. 12 (1966), 1901-1909.

Fuller, S., Siewiorek, D. and Swan, R., "Computer Modules: An Architecture for Large Digital Modules," ACM/IEEE First Annual Symposium on Computer Architecture, Gainesville, Fla. (Dec. 1973), 231-239.

Gordon, G., ***System Simulation***, Prentice-Hall, Englewood Cliffs, N.J. (1969).

Gosden, J.A., "Explicit Parallel Processing Description and Control in Programs for Multi- and Uni-processor Computing," ***FJCC*** (1966), 651-660.

Graham, R.L., "Bounds on Multiprocessing Anomalies and Related Packing Algorithms," ***SJCC*** (1972), 205-217.

Heart, F.E., Ornstein, S.M., Crowther, W.R. and Barker, W.B., "A New Minicomputer/multiprocessor for the ARPA Network," ***Proc. NCC*** (1973), 529-537.

Holt, A. and Commoner, F., "Events and Conditions," in ***Concurrent Systems and Parallel Computation Conference***, ACM (1970), , 1-52.

Holt, R.C., "Some Deadlock Properties of Computer Systems," ***Computing Surveys*** Vol. 4 No. 3 (Sept. 1972), 179-196.

Horning, J.J. and Randell, B., "Process Structuring," ***Computing Surveys*** Vol. 5 No. 1 (Mar. 1973), 5-30.

Howard, R.L., ***Dynamic Probabilistic Systems***, Vol. I: ***Markov Models*** and Vol. II: ***Semi-Markov and Decision Processes***, John Wiley & Sons, Inc., New York (1971).

Johnsson, R.K., "An Approach to Global Register Allocation," Ph. D. Thesis, Computer Science Dept., Carnegie-Mellon University (1975).

Karp, R.M. and Miller, R.E., "Properties of a Model for Parallel Computations, Determinancy, Termination, Queueing," ***SIAM J. Appl. Math.*** Vol. 14 No. 6 (Nov. 1966), 1390-1411.

Karp, R.M. and Miller, R.E., "Parallel Program Schemata," ***Journal of Computer and System Sciences*** Vol. 3 (1969), 147-195.

Keller, R.M., "Parallel Program Schemata and Maximal Parallelism I: Fundamental Results," *JACM* Vol 20 No. 3 (1973) (a), 514-537.

Keller, R.M., "Parallel Program Schemata and Maximal Parallelism II: Construction of Closure," *JACM* Vol 20 No. 4 (1973) (b), 696-710.

Kleinrock, L., *Queueing Systems Volume I: Theory*, John Wiley & Sons (1975).

Lehman, M., "A Survey of Problems and Preliminary Results Concerning Parallel Processing and Parallel Processors," *Proc. IEEE* Vol 54 No. 12 (Dec. 1966), 1889-1901.

Lesser, V.R., "The Design of an Emulator for a Parallel Machine Language," PhD Thesis, Elect. Eng. Dept., Stanford University (1972).

Lesser, V.R., Fennell, R.D., Erman, L.D. and Reddy, D.R., "Organization of the Hearsay II Speech Understanding System," *IEEE Symposium on Speech Recognition* (Apr. 1974), 11-22.

Levin, R., Cohen, E., Jefferson, D., Pollack, F. and Wulf, W., *HYDRA User's Manual*, Carnegie-Mellon U. Report (prelim), 1975.

Lunde, A., "POOMAS, Poor Man's Simula," unpublished report, Computer Science Dept., Carnegie-Mellon University (1971).

Martin, D.F. and Estrin, G., "Models of Computational Systems -- Cyclic to Acyclic Graph Transformations," *IEEE Trans. Comput.* EC-10 No. 1 (1967), 70-79.

Martin, D.F. and Estrin, G., "Path Length Computations on Graph Models of Computation," *IEEE Trans. Comput.* C-18 (1969), 530-536.

McMillan, C. and Gonzalez, R., *Systems Analysis: a Computer Approach to Decision Models*, Richard D. Irwin, Inc. (1968).

Merlin, P.M., "A Note on Recoverability of Modular Systems," Dept. of Information and Computer Science, University of California, Irvine (1975).

Miller, R.E., "A Comparison of Some Theoretical Models of Parallel Computation," *IEEE Trans. Comput.* C-22 No. 8 (Aug. 1973), 710-717.

Mills, H.D., "Top-down Programming in Large Systems," in *Debugging Techniques in Large Systems*, R. Rustin, Ed., Prentice-Hall, Englewood Cliffs, N.J. (1971), 41-55.

Miranker, W.L., "A Survey of Parallelism in Numerical Analysis," *SIAM Review* Vol 13 No. 4 (Oct. 1971), 524-547.

Newell, A. et al., *Speech-Understanding Systems: Final Report of a Study Group*, Computer Science Departmental Report, Carnegie-Mellon University (May 1971).

Newell, A. and Robertson, G., "Some Issues in Programming Multi-mini-processors," Computer Science Departmental Report, Carnegie-Mellon University (Jan. 1975).

Noe, J. and Nutt, G., "Macro E-nets for Representation of Parallel Systems," ***IEEE Trans. Comput.*** C-22 No. 8 (Aug 1973), 718-727.

Parnas, D., "On the Criteria to be used in Decomposing Systems into Modules," Computer Science Departmental Report, Carnegie-Mellon University (Aug. 1971).

Parnas, D., "A Technique for Software Module Specification with Examples," ***CACM*** Vol. 15 No. 5 (May 1972), 330-336.

Parnas, D., "The Influence of Software Structure on Reliability," ***Proc. Inter. Conf. on Reliable Software***, IEEE (1975), 358-362.

Paterson, M.S. and Hewitt, C.E., "Comparative Schematology," In ***Concurrent Systems and Parallel Computation Conference***, ACM (1970), 119-127.

Petri, C.A., "Kommunication mit automaten," Translated in Project MAC-M-212 Report, Originally published in 1962.

Prosser, R.T., "Applications of Boolean Matrices to the Analysis of Flow Diagrams," ***Proc. of the Eastern Joint Computer Conference*** (1959), 133-138.

Quatse, J.T., Gaulene, P. and Doge, D., "The External Access Network of a Modular Computer System," ***Proc. SJCC*** (1972), 783-790.

Regis, R.C., "Systems of Concurrent Processes: Structural Properties and Stochastic Analysis," Computer Science Dept., Johns Hopkins University (1972).

Rice, D.R., "An Analytical Model for Computer System Performance Evaluation," ***SIGME*** Vol. 2 No. 2 (June 1973), 14-30.

Riddle, W., "The Modeling and Analysis of Supervisory Systems," Ph.D. Thesis, Elect. Eng. Dept., Stanford University (1972).

Rodriguez, J.E., "A Graph Model for Parallel Computation," Ph.D. thesis, Elect. Eng. Dept., Massachusetts Institute of Technology (1967).

Rosenfeld, J., "A Case Study in Programming for Parallel-Processors," ***CACM*** Vol. 12 No. 12 (Dec. 1969), 645-655.

Simon, H., "The Architecture of Complexity," ***Proc. of the Amer. Phil. Soc.*** Vol. 106 No. 6 (Dec. 1962), 467-482.

VanLehn, K.A., Ed., ***Sail User Manual***, Stanford Artificial Intelligence Laboratory, Memo AIM-204, 1973.

Warshall, S., "A T...orem on Boolean Matrices," *JACM* Vol 9 No. 1 (1962), 11-12.

Weinberg, G.M., *The Psychology of Computer Programming*, Van Nostrand Reinhold Co., New York (1971).

Wulf, W., Russell, D. and Habermann, A., "Bliss: A Language for Systems Programming," *CACM* 14 (December 1971), 780-790.

Wulf, W. et al., *Bliss-11 Programmers Manual*, Digital Equipment Corporation, Maynard, Mass. (1972) (a).

Wulf, W. and Bell, C.G., "C.mmp -- a Multi-mini-processor," *Proc. AFIPS 1972 FJCC* Vol. 41, AFIPS Press, Montvale, N.J.(1972) (b), 765 - 777.

Wulf, W., Cohen, E., Corwin, W., Jones, A., Levin, R., Pierson, C. and Pollack, F., "HYDRA: The Kernel of a Multiprocessor Operating System," *CACM* Vol. 17 No. 6 (1974), 337-345.

Wulf, W. et al., *The Design of an Optimizing Compiler*, American Elsevier (1975) (a).

Wulf, W., Levin, R. Pierson, C., "Overview of the HYDRA Operating System Development," Proceedings of 5th Symposium on Operating Systems Principles, ACM SIGOPS, Austin, Texas, (Nov. 1975) (b), 122-131.

INDEX

adjacent processes, IV-15
adjacent, II-10
APPLY, A-9
attached to, II-10
ATTRIBUTE, A-10

BLACK BOX, IV-24

chains, IV-2
CLEAR, A-11
closed loop, IV-10
closed path, IV-12
COLLECT, A-11
combine two states, IV-19
Comments, A-3
connecting nodes, A-4
CONTINUE, A-11
COPY, A-12

deadlock problem, IV-8
deadlock-free, IV-14
delay time, II-5
DELAY, IV-15
DENSITY, A-14
DISPLAY, A-15

eliminate a state, IV-20
EXIT, A-16

Graph Reduction Process, IV-14
Graph Reduction Theorem, IV-29
graph reductions, IV-14

immediate-recurrent, II-10
in-parallel, IV-14
in-sequence, II-10
Irreducibility Theorem, IV-28

keyword abbreviations, A-2

Lemma R1.1, IV-16
Lemma R1.2, IV-16
Lemma R1.3, IV-17
Lemma R1.4, IV-17
Lemma R1.5, IV-17
Lemma R2.1, IV-19
Lemma R2.2, IV-20
line continuation, A-3
LINK, I-17, I-19
links, II-14
LOAD, A-16
loop, IV-10

Markov assumption, IV-1
Markov process, IV-1
message, II-3
MODEL, A-16
monodesmic process, IV-2

node, II-9

one-to-one, II-10
onto, II-10

path, II-10
ports, II-1
PROCESS, I-19
process, II-1

R1, IV-13,IV-15
R1a, IV-15
R1b, IV-15
R2, IV-14
R2a, IV-14
R2b, IV-14
R2c, IV-14
R3, IV-14, IV-24
R4, IV-14, IV-26
Reducibility Theorem, IV-26
REMOVE, A-17

safe, IV-8
SAVE, A-17
SCHEDULE, A-18
semi-Markov process, IV-1, IV-4
SIMULATE, A-18
SINK, IV-11
SNAPSHOTS, A-19

SOURCE, IV-11
SOURCE/SINK, IV-21
spaces, A-2
split paths, IV-12
start-up time, II-5
STATISTICS, A-19
steady state, IV-2
successor states, IV-16

TEST, A-20
Theorem R1, IV-17
Theorem R2, IV-22
Theorem R3, IV-25
Theorem R4, IV-26
transient states, IV-2
transition matrix, I-21
transition relation matrix, IV-7

UNSIMULATE, A-20

well-formed criteria, IV-6
well-formed graph, IV-6, IV-7
well-formed model, I-25,IV-6
well-formed process, IV-6