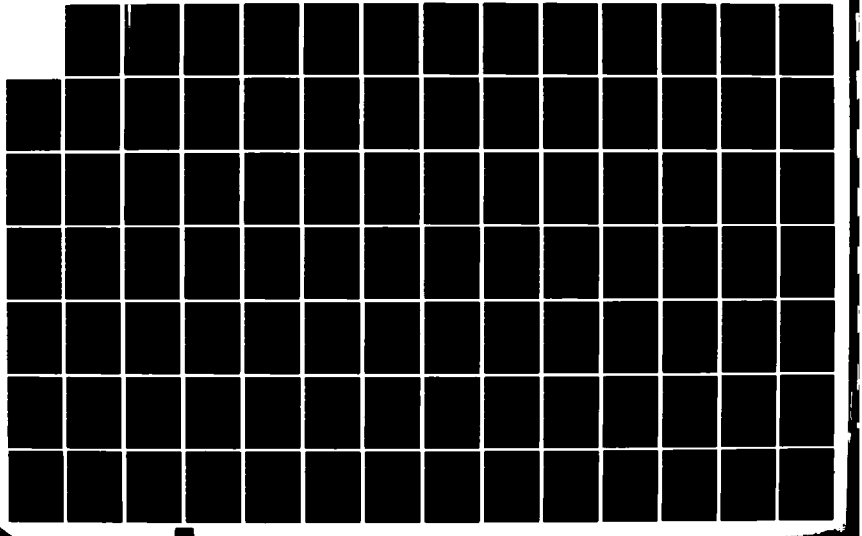
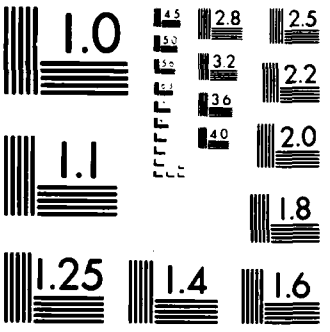
AD-A150 009 VERY-HIGH LEVEL CONCURRENT PROGRAMMING(U) MOORE SCHOOL OF ELECTRICAL ENGINEERING PHILADELPHIA PA DEPT OF COMPUTER AND INFORMATION SCIENCES Y SHI DEC 84 1/3

UNCLASSIFIED N00014-83-K-0560 F/G 9/2 NL





MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS 1963-A

REPRODUCED AT GOVERNMENT EXPENSE

AD-A150 009

VERY-HIGH LEVEL CONCURRENT PROGRAMMING

December 1984

by

Yuan Shi



DTIC
SELECTE
JAN 22 1985
E

UNIVERSITY of PENNSYLVANIA

The Moore School of Electrical Engineering

PHILADELPHIA, PENNSYLVANIA 19104

DTIC FILE COPY

This document has been approved for public release and sale; its distribution is unlimited.

85 01 22 052

2

University of Pennsylvania
Department of Computer and Information Science
Moore School of Electrical Engineering
Philadelphia, Pennsylvania 19104

VERY-HIGH LEVEL CONCURRENT PROGRAMMING

December 1984

by

Yuan Shi

Submitted to
Information System Program
Office of Naval Research
Under Contract N00014-83-K-0560

Moore School Report

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

REPORT DOCUMENTATION PAGE	READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER	2. GOVT ACCESSION NO. 12-A150 C09 / 3. RECIPIENT'S CATALOG NUMBER
4. TITLE *(and Subtitle)* VERY-HIGH LEVEL CONCURRENT PROGRAMMING	5. TYPE OF REPORT & PERIOD COVERED Technical Report
	6. PERFORMING ORG. REPORT NUMBER Moore School Report
7. AUTHOR(s) Yuan Shi	8. CONTRACT OR GRANT NUMBER(s) N00014-83-K-0560
9. PERFORMING ORGANIZATION NAME AND ADDRESS University of Pennsylvania, Moore School of Electrical Engineering, Department of Computer Science, Philadelphia, Pennsylvania 19104	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS	12. REPORT DATE DECEMBER 1984
	13. NUMBER OF PAGES 255 pages
14. MONITORING AGENCY NAME & ADDRESS*(if different from Controlling Office)*	15. SECURITY CLASS. *(of this report)* UNCLASSIFIED
	15a. DECLASSIFICATION/DOWNGRADING SCHEDULE
16. DISTRIBUTION STATEMENT *(of this Report)* ~~General Distribution~~	
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)* Concurrent programming, distributed computing, automatic program generation, system configuration, distributed termination control, distributed solution to simultaneous equations.	
20. ABSTRACT *(Continue on reverse side if necessary and identify by block number)* Enclosed	

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE
S/N 0102-014-6601

SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

VERY-HIGH LEVEL CONCURRENT PROGRAMMING

Yuan Shi

A DISSERTATION

in

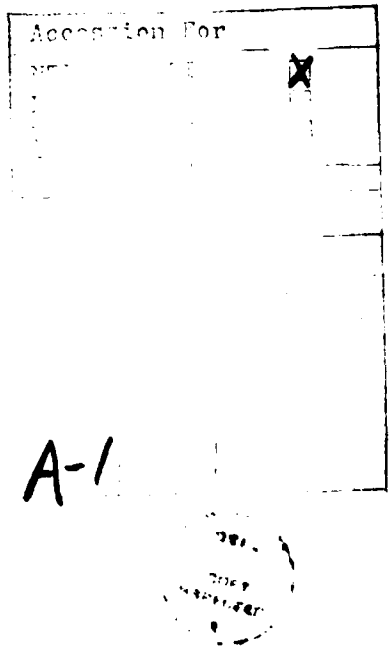
Computer and Information Science

Presented to the Graduate Faculties of the University of Pennsylvania in Partial Fulfillment of the Requirements for the Degree of the Doctor of Philosophy.

1984

Supervisor of Dissertation

Graduate Group Chairperson



ACKNOWLEDGEMENT

I wish to express my sincere gratitude to Dr. Noah Prywes, for his supervison and review of this dissertation. I am also grateful to my committee members, Drs. Timothy Finin, Boleslaw Szymanski, Eva Yuen-Wah Ma, Insup Lee and Peter Buneman, for their critical reviews and suggestions, and to Dr. Amir Pnueli for the valuable discussions and inspiring hints during the early stage of the work. I am indebted to Dr. Boleslaw Szymanski for his efforts and time spent on shaping up the dissertation.

I would also like to thank Jine Tseng, Peter Kaplan and Evan Lock for their use of the MODEL compiler and the Configurator. It is their "wild use" that made the current versions of the two systems much "cleaner". I thank Katherine Need, Spiros Pissinos, Sabiha Khalil and Lap-Ming Liu for proof reading various parts of the dissertation.

I am also indebted to an American and a Chinese family, the first of whom, Mr. and Mrs Gwinn, who have given me the unique opportunity to improve my English and facilitated my adjustment in American life. I thank them for their kindness and support for my study. I would also like to thank the members of my Chinese family in Shanghai. Their moral support and encouragement have contributed to the realization of my academic pursuits.

Finally, the support from ONR is deeply appreciated.

ABSTRACT

VERY HIGH-LEVEL CONCURRENT PROGRAMMING

Yuan Shi

Supervisor: Noah Prywes

Concurrent systems are typically large and complex, requiring long development time and much labor. They are, therefore, prime candidates for simplification and automation of the design and programming process. Their major application areas include real time systems, operating systems and cooperative computation by a number of independently developed geographically dispersed subsystems. New applications are emerging with the trends towards wide usage of personal computers connected in a network and towards use of parallel processing in supercomputer architectures.

The prime contribution of this dissertation is the creation of a programming style and an environment that allows the human designers to develop an implementation independent, very high level functional view of the concurrent systems. The translation of this view into a concurrently operating system is performed automatically. There is an emphasis on the human engineering aspects of the designer - computer interactions. The designers specify the problem through declaring variable structures and composing equations which relate the variables. Thus the specification is entirely declarative and assertive, without reference to its computerization. The designers partition the overall specification into modules which are each defined independently. These modules also become candidates for being computed concurrently. Each module consists of a subset of the variable declarations and equations. The designers view the concurrent system statically, as if all input and output data are available a priori, and the equations provide mathematical relationships among the data. The semantics of submitting the

specification to the computer is to have the computer give appropriate values to variables that all the equations are true. Excluded are such dynamic implementation concepts as sequences of program events, synchronization, exchanges of messages and relative timing. To accommodate the large size of typical systems, the methodology supports independence in specifying and testing individual modules. To aid debugging and attain reliability, language processors detect inconsistency and incompleteness errors in both the individual modules and in the global system. The translation from the specification into a respective computation by an object computer architecture is performed by the language processors. The entire design, prototyping and simulation of a system can be performed on a host computer and eventually moved to an object distributed computer system which is put into productive operation.

The dissertation describes an investigation of this approach using as the object computer architecture a modern distributed system consisting of interconnected sequential processes, each operating under a multiprogramming timesharing operating system.

The designers of a concurrent system interact with automatic systems on two levels: On the global level, the Configurator accepts as input a graph of the network of subsystems, modules and files. It verifies the validity of interfaces and implements the network by generating command language programs that set up communications and optimize parallelism among modules. The modules are executed under multiprogramming time sharing operating systems in respective sequential processors in a network.

On the local level, the MODEL Compiler accepts as input an individual module specification. It performs checking of completeness and consistency of variables and equations and generates an optimized sequential program in a high level language (PL/I).

The above two systems interact in checking the integrity of the specified system and generating the implementation programs. They have been implemented in PL/I, in the environment of Digital's VAX/VMS

operating system. Thus, automatic program design and generation methodology is used to translate the very high level specification into an efficient customized concurrent computation in a chosen environment.

One contribution of the dissertation of the dissertation is the exploration of the generality and power of this style of application systems development. This style of programming is novel and there has been little experience with it. The overall methodology is illustrated through two characteristic examples: a resource allocator, widely used in real-time systems, and a cooperative development of econometric models in a distributed environment. The examples present the new style of programming.

The other contributions of the dissertation are in the solutions to specific concurrent system design problems. This consisted of employing new concepts and algorithms. The implementation of a specification is based on communication of messages among concurrent processes. This requires checks of the specification to alert the designer to the existence of inconsistencies and automatic design of implicit synchronization and prevention of deadlocks. The entire concurrent system must cooperate in the distributed computations, especially in initiation and termination of system-wide iterative computations.

The dissertation consists of three parts. Part I presents the new style of specifying concurrent systems, as well as high level descriptions of automatic design and programming environment. Part II documents the design of the Configurator. part III documents the modifications to the previously developed MODEL compiler which were necessary for concurrent operation of programs and communications among the programs.

TABLE OF CONTENTS

PART I. OVERVIEW OF THE METHODOLOGY FOR HIGH-LEVEL CONCURRENT PROGRAMMING

CHAPTER 1 INTRODUCTION

1.1 THE PROBLEM AND THE OVERALL APPROACH 2
1.2 CONTRIBUTIONS 5
1.3 PRINCIPAL CHARACTERISTICS OF THE VERY-HIGH LEVEL LANGUAGE 6
1.4 PROGRAM DEVELOPMENT PROCEDURE 8
1.5 ASSUMPTIONS 10
1.6 USE OF EXAMPLES 11
1.6.1 RESOURCE ALLOCATION 11
1.6.2 COOPERATIVE PROGRAMMING 12
1.7 RELATED WORK 14
1.7.1 CONCURRENT PROLOG 14
1.7.2 MODULARITY IN UNIX 15
1.7.3 DATAFLOW MACHINES 15
1.7.4 SURVEY OF OTHER CONCURRENT PROGRAMMING LANGUAGES 15
1.8 ORGANIZATION OF THE DISSERTATION 19

CHAPTER 2 COMPOSING A CONFIGURATION OF MODULES AND FILES

2.1 MODULES AND FILES 21
2.2 CONFIGURATION OF THE DINING PHILOSOPHER EXAMPLE . 21

CHAPTER 3 OPERATION OF THE CONFIGURATOR

3.1 FUNCTIONS AND PHASES OF THE CONFIGURATOR 27
3.2 CHECKING 30
3.2.1 COMPLETENESS AND INTER-MODULE CONNECTIONS . . . 30
3.2.2 CONSISTENCY OF TEMPORAL RELATIONS 30
3.2.3 COMPATIBILITY OF INTERFACING FILE DESCRIPTIONS . 31
3.3 OPTIMIZATION 31
3.4 DIAMETER EVALUATION 32
3.5 CODE GENERATION 32

CHAPTER 4 SPECIFYING INDIVIDUAL MODULES - RESOURCE ALLOCATOR

4.1 THE PHILOSOPHER MODULE 33
4.2 THE RESOURCE ALLOCATOR MODULE R. 38

CHAPTER 5 THE OPERATION OF THE MODEL COMPILER

5.1 REPRESENTATION OF THE SPECIFICATION AS AN ARRAY GRAPH 46
5.2 CHECKING COMPLETENESS AND CONSISTENCY OF A SPECIFICATION 47
5.3 OPTIMIZATION OF PRODUCED PROGRAMS 48
5.4 EXTENSIONS FOR CONCURRENT PROCESSING 50
5.4.1 EXTERNAL DEPENDENCY 50
5.4.2 THE MAIL AND POST FILES 51
5.4.3 CONCURRENT ISAM FILE UPDATES 51
5.4.4 ITERATIVE SOLUTION TO DISTRIBUTED SIMULTANEOUS EQUATIONS 51

CHAPTER 6 A SECOND EXAMPLE - COOPERATIVE COMPUTATION

6.1 SPECIFICATION OF INDIVIDUAL ECONOMETRIC MODULES . 53
6.2 CONFIGURING A MULTI-ECONOMETRIC MODEL SYSTEM . . . 56

CHAPTER 7 DISTRIBUTED SOLUTION OF SIMULTANEOUS EQUATIONS

CHAPTER 8 CONCLUSION AND FUTURE RESEARCH

PART II. THE CONFIGURATION SYSTEM

CHAPTER 9 INTRODUCTION TO THE CONFIGURATOR

9.1 THE LANGUAGE - CSL 66
9.2 THE PROCESSOR - CONFIGURATOR 67

CHAPTER 10 THE CONFIGURATION SPECIFICATION LANGUAGE

10.1 OVERALL GRAPH DESCRIPTION 68
10.2 NODES IN A PATH STATEMENT 71
10.3 MODULE NODE ATTRIBUTES 71
10.3.1 MODULE TYPE ATTRIBUTE (M_TYPE) 72
10.3.2 PHYSICAL NAME ATTRIBUTE (PHYSICAL_NAME) 73
10.4 FILE NODE ATTRIBUTES 74
10.4.1 FILE TYPE ATTRIBUTE (F_TYPE) 74
10.4.2 PHYSICAL NAME ATTRIBUTE (PHYSICAL_NAME) 77
10.4.3 RECORD SIZE ATTRIBUTE (RECORD_SIZE) 77
10.4.4 FILE DEVICE ATTRIBUTE (F_DEVICE) 77
10.5 EDGES 78
10.6 REQUIREMENTS OF CONNECTING FILES 79
10.6.1 CONCURRENCY, DISTRIBUTIVITY AND SEQUENTIALITY . 79
10.6.2 COMPATIBILITY OF CONNECTING FILES 80
10.6.3 MANUAL INITIATION OF MODULES 82
10.7 SYNONYM STATEMENT 82
10.8 OPERATING THE CONFIGURATOR 83
10.8.1 INVOKING THE CONFIGURATOR 83
10.8.2 I/O FILE NAMING CONVENTIONS 83
10.8.3 PARAMETERS TO THE CONFIGURATOR 84

CHAPTER 11 THE CONFIGURATOR

11.1 STRUCTURE 85
11.2 PRODUCTS 87
11.3 THE MAIN CONFIGURATOR PROGRAM: CONF 88
11.4 SYNTAX ANALYSIS AND CONFIGURATION GRAPH CONSTRUCTION (PROCEDURE NAME: SAP) 88
11.4.1 THE SYNTAX ANALYZER 88
11.4.2 DEFINITION OF A CONFIGURATION GRAPH 91
11.4.3 CONSTRUCTION OF THE CONFIGURATION GRAPH 92
11.4.3.1 DATA STRUCTURES USED IN THE GRAPH CONSTRUCTION . 97
11.5 COMPLETENESS ANALYSIS (PROCEDURE NAME: CMPANA) . . 98
11.6 SEQUENCE CHECKING (PROCEDURE NAME: SEQCK) 99
11.6.1 PRELIMINARIES 99
11.6.1.1 REQUIREMENT 99
11.6.1.2 TEMPORAL RELATIONS 101
11.6.2 SEQUENCING ANALYSIS 103
11.6.2.1 CIRCULAR FORM INCONSISTENCY 104
11.6.2.2 NON-CIRCULAR FORM INCONSISTENCY 107
11.7 SCHEDULING (PROCEDURE NAME: SCHEDULE) 111
11.7.1 MORE CONSISTENCY ANALYSIS 111

11.7.2 SCHEDULING 113
11.8 MSCC DIAMETER EVALUATION (PROCEDURE NAME: FDMAX) . 115
11.9 CODE GENERATION (PROCEDURE NAME: GCODE) 118
11.9.1 MAIN JCL PROGRAM GENERATION 120
11.9.2 INDIVIDUAL JCL PROGRAM GENERATION 121
11.9.3 PL/I PROGRAM GENERATION 123
11.10 CONFIGURATION DOCUMENTATION (PROCEDURE NAME: GRPT) 124

PART III. MODIFICATIONS TO THE MODEL COMPILER

CHAPTER 12 OBJECTIVES OF THE MODIFICATIONS

CHAPTER 13 EXTERNAL DEPENDENCY

13.1 FUNCTION OF EXTERNAL DEPENDENCY 131
13.2 SYNTAX ANALYSIS 132
13.3 ARRAY GRAPH ANALYSIS (PRECEDENCE ANALYSIS) 132
13.4 RANGE AND DATA TYPE PROPAGATION 132
13.5 CODE GENERATION 133

CHAPTER 14 THE MAIL AND POST FILES

14.1 FUNCTION 134
14.2 SYNTAX ANALYSIS 134
14.3 CODE GENERATION 136
14.3.1 THE OPEN PROCESS 136
14.3.2 THE CLOSE PROCESS 138
14.3.3 THE READ PROCESS 138
14.3.4 THE WRITE PROCESS 139
14.3.5 ON ENDFILE 140

CHAPTER 15 CONCURRENT UPDATE OF ISAM FILES

15.1 PROBLEMS AND OBJECTIVES 142
15.2 SCHEDULING 143
15.3 CODE GENERATION 144

CHAPTER 16 ITERATIVE SOLUTION FOR DISTRIBUTED SIMULTANEOUS EQUATIONS

16.1 PROBLEMS AND OBJECTIVES 146
16.2 SOLUTION TO DISTRIBUTED TERMINATION PROBLEM . . . 147
16.2.1 COMPARISON WITH THE KNOWN SOLUTIONS 147
16.2.2 ASSUMPTIONS OF THE TERMINATION CONTROL ALGORITHM 147
16.2.3 THE TERMINATION ALGORITHM AND ITS DERIVATION . . 148
16.2.4 FINDING THE DIAMETER OF THE NETWORK DYNAMICALLY 152
16.3 GENERAL DESCRIPTION 156
16.4 SCHEDULING 157
16.5 THE PROCEDURE "PREPARE" 163
16.6 CODE GENERATION 167
16.6.1 ACQUIRING THE DIAMETER OF A NETWORK 167
16.6.2 ATTACHMENT OF THE TERMINATION CONTROL ALGORITHM 167

APPENDIX A. EXAMPLES
A1. THE RESOURCE ALLOCATION EXAMPLE 171
A1.1 THE PHILOSOPHER MODULE 171
A1.2 THE R MODULE 177
A1.3 CONFIGURATION DOCUMENTATION. 188

A1.4 SIMULATION REPORT. 194
A2. THE COOPERATIVE COMPUTATION EXAMPLE 195
A2.1 SYSTEM CONFIGURATION. 195
A2.2 THE WORLD MODULE. 197
A2.3 THE USA MODULE. 210
A2.4 THE JAPAN MODULE. 213
A2.5 THE TAIWAN MODULE 216
A2.6 THE KOREA MODULE. 219
A2.7 THE PHILIPPINE MODULE 222
A2.8 THE THAILAND MODULE 225
A2.9 SIMULATION RESULTS. 228
A2.10 EXECUTION STATISTICS. 238

APPENDIX B
EBNF OF CONCURRENT MODEL. 239

APPENDIX C. WARNING AND ERROR MESSAGES GENERATED BY THE CONFIGURATOR
C1. ERROR/WARNING MESSAGES FOR SYNTAX ANALYSIS 244
C2. ERROR/WARNING MESSAGES FOR CONFIGURATION GRAPH CONSTRUCTION 244
C3. ERROR/WARNING MESSAGES DURING COMPLETENESS ANALYSIS. .245
C4. ERROR/WARNING MESSAGES DURING SCHEDULING AND DOCUMENTATION GENERATION 246

BIBLIOGRAPH. 248

INDEX. 253

LIST OF FIGURES

FIGURE 1. Schematic Diagram of Concurrent Programming Procedure. . 8
FIGURE 2. Configuration Network For Reusable Resources Allocation Example. 22
FIGURE 3. Specification of Configuration of Figure 2 25
FIGURE 4. Specification of Philosopher Module. 34
FIGURE 5. External Dependency As Viewed From The Philosopher Module 36
FIGURE 6(a). Resource Allocator Module Specification: Header, Data Description and Definition of Data Parameters. . 39
FIGURE 6(b). Resource Allocator Module Specification: Equations For Variables in Files QUEUE and ALLOC and the external dependencies 42
FIGURE 6(c). Simulation Report Specification of the Resource Allocation Module. 44
FIGURE 7. Illustration of External Dependencies As Viewed From the R Module 45
FIGURE 8. A Reduced Example of a Local Econometric Model 54
FIGURE 9. Specification of a Reduced Econometric Model of a Country. 55
FIGURE 10(a). Configuration Network For Pacific Basin Model . . . 57
FIGURE 10(b). Configuration Specification of Pacific Basin Model . 58
FIGURE 11. Syntax structure of a CSL specification 68
FIGURE 12. Syntax of a path statement. 69
FIGURE 13. Syntax of a Synonym Statement 70
FIGURE 14. Syntax of a configuration node. 71
FIGURE 15. Syntax of an attribute of a module 71
FIGURE 16. Syntax of attribute description of a file node. 74
FIGURE 17. The Processing Stages of The Configurator 86
FIGURE 18. EBNF/WSC Description of CSL 90
FIGURE 19. Segment EBNF/WSC of the Path Statement. 93
FIGURE 20. Segment SAP of the Path Statement 94
FIGURE 21. Two possible sub-configurations of a GRP module 100
FIGURE 22. Execution of a Configuration System 120
FIGURE 23. Processing stages in MODEL compiler 130
FIGURE 24. Syntax of a file declaration in MODEL 134
FIGURE 25. PL/I code for concurrent ISAM file accessing. 145
FIGURE 26. The ON-UNIT for concurrent ISAM file accessing. 145
FIGURE 27. Cutting an edge for a MSCC enclosing a DEPENDS_ON statement 159
FIGURE 28. MSCCs in an array graph 162
FIGURE 29. Data structures of a flowchart. 164
FIGURE 30. Representation of a flowchart 165
FIGURE 31. Representation of a file structure and the patched token field 166
FIGURE 32. PL/I code for acquiring network diameter. 167
FIGURE 33. The recalculation procedure for simultaneous equations 168
FIGURE 34. The recalculation procedure with termination control. .169

LIST OF TABLES

TABLE 1. File and Module Production and Consumption Rules. 79
TABLE 2. I/O File Name Conventions of the Configurator 83
TABLE 3. Parameters to the Configurator 84
TABLE 4. Semantic Routines For CSL 91
Table 5. Transitivity table of Tempotral Relations 102
TABLE 6. Interpretations of a Source File node (OPEN) 137
TABLE 7. Interpretations of a Target File Node (CLOSE) 138
TABLE 8. Interpretations of a Source Record Node(READ) 139
TABLE 9. Interpretations of a Target Record Node(WRITE). 139

LIST OF ALGORITHMS

Algorithm 1. Vc construction (BVC). 105
Algorithm 2. Construction of Ec (BEC) 106
Algorithm 3. Report Sequencing Error (SEQERR) 107
Algorithm 4. Solving SFS problems in Gf (SLVSFS). 108
Algorithm 5. Construction of Ve (BVE) 113
Algorithm 6. Construction of Ee (BEE) 113
Algorithm 7. Report Inconsistency Error (CSTERR). 114
Algorithm 8. Finding diameter of a MSCC (FDMAX) 116
Algorithm 9. Generating main JCL programs (GCODE) 120
Algorithm 10. Producing individual JCL programs (GCODE). 122
Algorithm 11. Mailbox creation (GCODE) 123
ALGORITHM 12. DISTRIBUTED TERMINATION CONTROL. 152
ALGORITHM 13. DISTRIBUTED DIAMETER EVALUATION FOR A NODE X 154
Algorithm 14. MSCC UNRAVELLING (SIMUL_BLK) 158
ALGORITHM 15. FINDING EXTERNAL RECORD NAMES IN A MSCC (GENERATE) .166

PART I

HIGH-LEVEL CONCURRENT PROGRAMMING

CHAPTER 1
INTRODUCTION

1.1 THE PROBLEM AND THE OVERALL APPROACH

Concurrent computation is widely used in operating systems and in real time systems. There are novel application areas, such as robotics, which require coordination of a number of activities. It is also increasingly used in distributed processing systems for cooperative computation by geographically dispersed users. The greatest potential for use of concurrent computation is in the emerging parallel computer architectures [Arvind,83; Dennis,80; Gard,82; Smith,78; Treleavan, 82]. Programming of concurrent computing has proven to be very complex and prone to errors. Experience indicates that it consumes enormous amount of time for program development and maintenance. The difficulties encountered lie partly in the large size of typical concurrent systems, but more importantly in the need for the programmer to take into account sensitive interactions between parallel streams of program events. For these reasons, making concurrent programming easier has received much attention. A number of programming languages in the style of conventional high level languages have been developed [Brinch, 78; Hoare, 78; Holt, 78; Milner, 80]. More recently a new type of language, variously called definitional, nonprocedural, logical or dataflow, has been proposed for use in the new parallel computer architectures [Ackerman, 82; McGraw, 82; Hoffman, 82; Arvind, 78; Ramamarithan, 83; Backus, 78; Shapiro, 83]. However, in these

languages, the programmer still needs to visualize the solution of the problem in terms of streams of data, computations by processors and communications among processes. This level is considered here as still too low.

A very high level approach to this problem is described in this dissertation. It involves use of a mathematical specification of a computation which does not require operational semantics. It consists only of declarations of structures of variables and equations that relate the variables. Such a specification is composed without regard to, or even knowledge of, the underlying implementation of the computation. The translation of a specification into a computation on an object computer architecture is performed automatically. The computer architecture selected here to demonstrate the approach is that of a modern distributed processing system consisting of interconnected sequential processors, each run under a multiprogramming timesharing operating system. The translation of the specification into concurrent programs which communicate with each other is performed by two translating systems: a Configurator which implements the global aspects through generating command language programs, and a MODEL Compiler which implements the local aspects through generating high level language programs (PL/I) for individual processes. The translators use the VAX/VMS operating system and communication facilities, as well as a conventional PL/I programming language compiler. The selection of the VAX/VMS environment has been purely for demonstrating the approach. The methodology should be equally applicable to other computer architecture.

The above two translating systems offer the user assistance in debugging and validating of the concurrent system. The verification of a concurrent system poses many theoretical and practical problems. Several specification languages and methods have been proposed for verification of concurrent systems [Zave, 84; Lauer, 79; Chen, 83; Parnas, 74; Pnueli, 79]. Such specifications would require in typical practical applications a large amount of labor. Composing

such a specification is also prone to making numerous errors. Also all-automatic verification is not possible and human analysis and assistance is necessary, requiring high level of expertise from the user. These features would negate our objective of reducing labor and user expertise. The approach here requires only declarations of data structures and definitions of variables by equations. Checks were progressively incorporated in the Configurator and MODEL Compiler, for increasingly complex types of errors. They consist of checks of compatibility of various attributes of data structures referred in equations and checks of respective dependencies. The specification is checked for consistency of use of data types, dimensionality of arrays and ranges of dimensions. Dependency checks include the completeness of definitions of variables and analysis of circularity of definitions. Also checks are conducted of some rules for allowed dependencies. The above checks have been incorporated in the two translating systems and their effectiveness was evaluated experimentally [Cheng, 83]. An important consideration in reducing the number of errors is that the user employs only the very-high-level view and thus avoids making errors in the implementation level. Also all corrections and modifications to specification are done in the Configurator or MODEL languages. The automatic translators employ a variety of scheduling and communication protocols embodied in operating systems and communications technology, which have been verified and of which the user need not even be aware.

For real time systems, another step is necessary. Typically, real time systems have timing constraints. To satisfy the time constraints, a designer may have to partition a module into several smaller ones based on the estimated execution time produced at compilation time. A system for obtaining the needed timing information, based on the module specifications, is being developed [Tseng, 83].

The dissertation describes a very-high level language for concurrent programming which is devoid of implementation aspects and

it explores the effectiveness of programming with such a language (assisted by the two translators). It also describes the operation of the two translators - the Configurator and MODEL compiler - with emphasis on

i) semantic checking of the very-high-level language input and assisting the user in its composition,

ii) optimization of the overall computation by use of parallelism to reduce overall execution time and by minimizing the use of main memory storage and computation time in individual processors.

1.2 CONTRIBUTIONS

The objectives and contributions of the dissertation are as follows.

i) devising a very-high level language for concurrent programming which is devoid of implementation aspects.

ii) exploring the effectiveness of programming with such a language (illustrated by two examples)

iii) devising, demonstrating and exploring the operation of translators of the very-high level languages into an implementation of the computation in the object computer architecture, with emphasis on

   a) semantic checking of the very-high-level language input and assisting the user in its composition,

   b) optimization of the overall computation by use of parallelism to reduce overall execution time and by minimizing the use of main memory storage and computation time in individual processors.

The dissertation endeavors to make two points:

i) that very high level definitional, nonprocedural, dataflow

languages can be used very effectively and naturally in concurrent programming, and

ii) that automatic design and program generation methodology can support program development and generate an efficient implementation of the concurrent system.

1.3 PRINCIPAL CHARACTERISTICS OF THE VERY-HIGH LEVEL LANGUAGE

The principal characteristics of the proposed programming style, which distinguish it from conventional programming are summarized below.

i) An overall specification is partitioned into modules. The. user prepares a specification for each module. A specification of a module consists of declarations of variable structures, equations that define output variables in terms of input variables, and declarations of external dependencies of input variables on output variables. An external dependency declared in one module indicates that a function is specified in detail in other modules. A user engaged in composing a specification has to state whether an external dependency exists, but does not have to know the detailed definitions involved in the dependency. Thus a specification of a module becomes independent of other modules.

ii) A variable in a specification may assume only one value. This is similar to the approach taken in mathematics. This means that all the values evaluated in the eventual computation procedural program are represented in the high level view by distinct variables. This allows the user to view all input, interim and output variables statically, as if they assumed values a priori, and helps to compose equations that express the relationships among the variables.

iii) Specification statements may be in an arbitrary order and there are no control statements, such as for input-output, iterations,

etc. The user visualizes the specification, not as a set of commands to be performed by a computer - as in conventional programming languages - but as a set of equations that should be made true, by finding the appropriate values for variables. Thus, every equation is an "invariant" assertion.

iv) The synthesis of modules into a global system is specified in a configuration language. Modules and files are assembled into a configuration by defining a dataflow-like graph - with modules, subsystems and files as the nodes and their input-output relationships as the edges. A configuration may itself be a subsystem represented by a node in a higher level configuration. The evaluation of a configuration means making all the modules true (meaning that all the equations in the respective module specifications are true). Thus a configuration is viewed as a static description of a computation, similar to individual modules.

v) The user is not concerned with optimizing efficiency of computations. The automatic translators incorporate optimization for efficiency. They examine the efficiency of a much larger number and variations of possible computation schedules than a human programmer could possibly conceive and consider. Further, the user would have to be highly expert in the object computer architecture in order to offer guidance on efficiency. The one exception to this approach is where the user determines the partition of the overall specification into modules which the translators may schedule concurrently.

Parallel execution of recursive functions have not been included in the translators described here. Because of the recent interest in parallel execution of recursive functions in artificial intelligence, an extension to the system for dynamic initiation of recursive definitions is considered for future research.

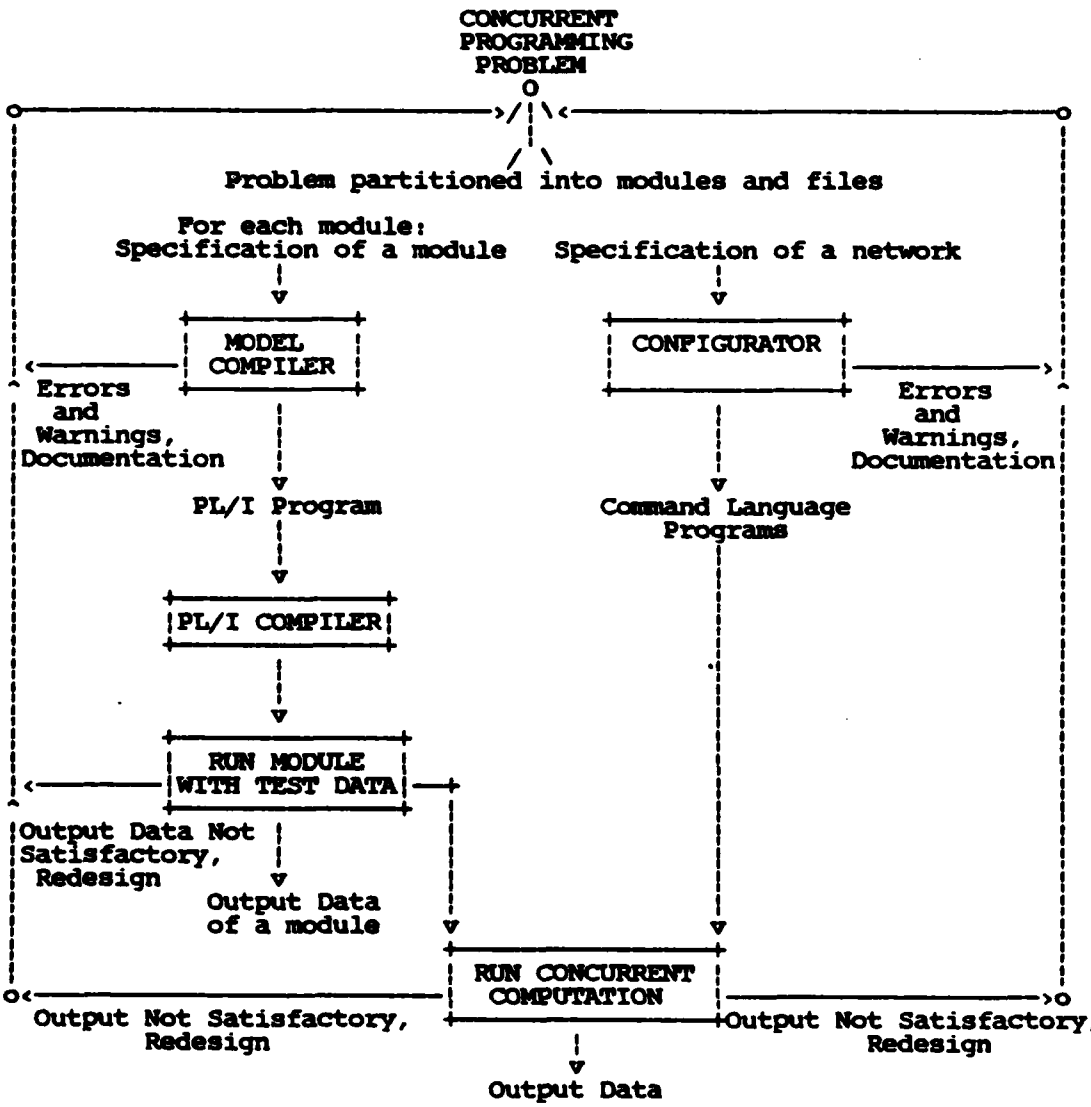


FIGURE 1. Schematic Diagram of Concurrent Programming Procedure

1.4 PROGRAM DEVELOPMENT PROCEDURE

The overall procedure in using this methodology is illustrated schematically in Figure 1. It starts (at the top) with existence of a concurrent programming problem. In the case of a top-down approach, the human users have to partition the problem into modules. In a bottom-up approach, the modules may already exist. There are two

parallel paths in Figure 1 for module definition and for global system synthesis. They merge at the bottom of the diagram to produce the concurrent computation. The order of employing these two paths depends on whether a top-down or a bottom-up approach is undertaken.

The path on the left is followed for each module in a configuration. In case of system modification, only the specifications of affected modules need to be added, deleted or changed. This path consists of composing a specification of a module in the MODEL language, and submitting it to the MODEL Compiler. The MODEL Compiler constructs a dataflow graph for the module specification. This graph is used for analysis of consistency and completeness of definitions, to discover errors, and for optimization of the generated program. The user must then make corrections to respond to error and warning messages issued by the MODEL Compiler. Finally, a program is generated, in our case in PL/I. The program can then be executed as a process, by itself for testing, and in concurrent operation with other modules as described in a configuration specification.

The path on the right of Figure 1 is used to integrate programs into a concurrent computation. A specification of a network of modules and files is submitted to the Configurator. The Configurator constructs a dataflow graph of the configuration and analyzes the graph for compatability of the interconnections and completeness. The user must make appropriate changes to respond to error or warning messages. The Configurator produces then an overall customized design to maximize the parallelism in execution of modules, and generates a set of command language programs for executing the network of modules in a chosen environment of computers, communications and their operating systems. The Configurator also performs system wide documentation, similar to previously developed systems [Teichreow, 77].

1.5 ASSUMPTIONS

A number of constraining assumptions ware made to permit the implementation of the Configurator and MODEL compiler using the existing hardware/software systems. They also define the application domain of the developed system.

The assumptions of the developed systems are listed as follows:

i) Hardware/Software Environment

The physical environment assumed is a computer network, where each node consists of one or more sequential computers that operate under multi-programming operating systems. The operating systems must have a file system for handling sequential, indexed sequential and mailbox files.

ii) Each module or file is assigned to a specific location in the computer network. Changes in location require re-specify the configuration.

iii) It is up to the user to define backup modules and files. Namely, the user must define backup files and recovery modules manually. The systems does not automatically incorporate such operations.

iv) No recursive module definition is allowed. This is restricted by the inability of dynamically creating modules. However, modules can be activated dynamically by addressing messages to them. Also, using the developed systems, i.e. the CONFIGURATOR and the concurrent MODEL compiler, recursion can be simulated iteratively.

The above assumptions implicitly restrict the scope of the research area and the application domain of the developed systems. For instance, dynamic module re-allocation is not addressed in this dissertation. Also, besides saving wrongly addressed messages, the

developed systems do not support failure recovery services automatically.

1.6 USE OF EXAMPLES

The style of programming here differs greatly from that of conventional procedural programming. The dissertation focuses on presenting the new style through two examples. The first example is of a resource allocator, such as found in operating systems or in real time systems. This example uses a top-down approach, where the overall system is partitioned first and then individual modules are specified. The second example illustrates cooperative computation in a distributed processing environment. It consists of econometric models for a group of countries that are linked together to form a regional econometric model. This example stresses a bottom-up approach - developing or modifying first the individual modules followed by their synthesis. The operation of the two translators is described only generally in the interest of brevity. The further detail of the examples are given in Appendix A.

1.6.1 RESOURCE ALLOCATION

Resource allocation captures the essence of many concurrent systems used in real-time applications. It is used in operating systems to allocate computing and input-output resources to jobs, and in real-time systems to allocate available resources to participants - such as routes and landing permissions in an air traffic control system. To simplify the example, only reusable resources are allocated. Allocation of consumable resources is illustrated in the second example. There are many strategies for allocating resources. The more complex ones use resources more efficiently and fairly while preventing a deadlock. Again for simplicity, the strategy selected

here avoids deadlocks by requiring that a module submit a maximum claim request for all the resources that it will need, and release them when not further needed. Also to satisfy the fairness requirement, requests are satisfied in strict order of arrival.

To make the example more specific and easier to follow it is stated in terms of the Dining Philosophers problem as related by [Hoare, 78] due to E. W. Dijkstra. This however does not restrict the generality of the example. Five philosophers share a circular dining table where each has an assigned seat. There is one fork between each two seats. A philosopher needs the forks to his right and left in order to dine. A philosopher desiring to dine requests the forks. When available, the resource allocator issues both forks and the philosopher proceeds to dine. When finished, he releases both forks, which become available to his immediate neighbors on a first-come-first-allocated basis.

1.6.2 COOPERATIVE PROGRAMMING

Concurrent programming has been considered in the past mainly as a top-down development process, outlining first the global aspects and then proceeding to fill in the local details. With the advent of computer technology, the price of computers has drastically declined and the computation power available in the past only in large scale "main-frames" has become available in small personal computers. This is bound to enhance connecting local computers to integrate many complementary computations which were developed independently. This mode of activity has been called cooperative computation. In this mode, definitions of local modules occur naturally reflecting the interest and expertise of local developers. The developers are typically initially uncoordinated and dispersed organizationally and geographically. The motivation for linking computers with modules and data into an integrated system comes later, based on recognition of

the interdependence of the respective problem areas. The advantage in synthesizing a global system may be viewed as follows. In an isolated module, the variables which are imposed by the external environment are considered as parameters and their values are assumed by the user. In contrast, in a global system these variables can be jointly evaluated, which makes the results much more reliable. The main difficulty in synthesizing a number of modules is frequently due to the difference in definitions given to essentially common variables in independently developed interacting modules. An agreement must be made between authors of such modules on needed transformations of these variables to obtain a common meaning and structure. Such an agreement is called a contract [Gana, 78] and is sometimes defined by adding an interfacing module which performs the translation.

Project LINK [Klein, 77] is a classical example of cooperative computation and is used here as an illustration. It consists of a number of institutions who have been developing stand alone econometric models, typically for their own country or region, and who cooperate in synthesizing their models into an area or world wide model. The databases and econometric equations in the local models are in constant flux due to political and economic changes. Since the respective economies are highly interdependent, it is very important to synthesize the models to evaluate the effect of the very latest developments. The synthesis of models is frequently performed on an ad-hoc basis. Also results must be obtained quickly to alert the decision makers to needed changes in economic policies and plans.

The second example has been proposed to us by Y. Yasuda of Kyoto University and of the staff of the LINK Project at the University of Pennsylvania. It consists of a study of economic interactions in the Pacific Basin. The economies studied and their corresponding models are those of the USA, Japan, Taiwan, Korea, Philippine and Thailand.

1.7 RELATED WORK

In proposing a nonprocedural approach for specification and implementation of a concurrent system, we are following in the footsteps of a number of proposed languages for concurrent programming.

The proposed language, however, is drastically different in its semantics from previously developed programming languages. It essentially requires composing data objects and their inter-relationship mathematically and the compiler will make all the definitions become true.

1.7.1 CONCURRENT PROLOG

PROLOG is a "tree" structured language. Each axiom is expressed and evaluated in a tree fashion with the answer at the top of the tree. In sequential PROLOG, the evaluation of the tree is depth-first and from left-to-right. The main idea of designing concurrent PROLOG is to explore the use of concurrency implied in "AND" and "OR" nodes in the tree. Each node in the tree can communicate with each other through passing messages. Since the message passing mechanism really bears the concept of dataflow, the concurrent PROLOG has a quite different programming style than sequential PROLOG, it has been called "object oriented programming" [E.Shapiro, 83].

MODEL is not a tree structure language. It uses the syntax similar to the notions used in algebra. Modules are defined by the user at a higher-level. The user of the MODEL language does not "see" messages passing between modules. He sees only differently organized entire files being produced and consumed by modules. The user also does not see the concurrency explicitly. It is up to the Configurator to decide the concurrency of the overall system. Of course, the more the modules are being partitioned, there are more candidates to be computed concurrently.

1.7.2 MODULARITY IN UNIX

Kernighan [84] points out the added dimension of modularity offered by connecting processes to form an integrated system (as an alternative to procedure modularity). He discussed the effectiveness of this mode of modularity when using UNIX. In UNIX, there is a mechanism called "pipeline" which can be used to construct bigger and complex programs by connecting small and simpler processes. A "pipeline" is a message channel between processes.

In MODEL, the devices similar to a "pipeline" are the MAIL and POST files. The MAIL and POST files offer much greater flexibility in communications between modules, including 1 to many and many to 1 distribution of messages. It therefore further enhances this mode of attaining modularity.

1.7.3 DATAFLOW MACHINES

Using the currently developed Configurator and the MODEL compiler, the concurrency of an application system is purely on the module level. It is based on the partitioning of the overall specified application system. There is no concurrency below the module level, because the MODEL compiler generates a sequential program for each module specification. This, however, is not a limitation of the proposed approach, the MODEL compiler could equally well produce parallelism within each module. Maya Gokhale [Gokhale,83] demonstrated how to directly translate a MODEL specification into MAD, a low-level dataflow language designed for the Manchester dataflow machine. Thus an integrated dataflow system is feasible by using the Configurator (at high-level) and Gokhale's MODEL compiler (at lower-level) to operate a cluster of dataflow machines concurrently.

1.7.4 SURVEY OF OTHER CONCURRENT PROGRAMMING LANGUAGES

Historically, concurrent programming languages used either message passing or shared memory for inter-module communication. This approach required analysis of the timing and waiting patterns of the global computation events. This contrasts with the use of files for inter-module communication which eliminates lower-level timing considerations.

*Concurrent Pascal [Hansen, 77]

A concurrent programming language based on the MONITOR concept and emphasizing structured programming for concurrent programs. More recently, a new version of concurrent Pascal (EDISON) [Hensen, 81] was developed. In EDISON, a new mechanism supporting abstract data type is implemented. Use of this language can produce more structured and modularized programs than its ancestor.

The MONITOR concept requires shared memory hardware and is not readily usable in distributed processing. However, the basic concept of a MONITOR for resource allocation can be expressed in MODEL as illustrated in the resource allocation example (Chapter 4).

*Communicating Sequential Processes (CSP) [Hoare, 78 and 81],

A concurrent programming language using message passing which combines the guarded command suggested by Disjkstra and parallel composition of processes. It uses only primitive message send and receive constructs. The intention of creating this language was to provide a formalism for concurrent programming.

*Concurrent SP/k (CSP/k) [Holt et al, 78]

An extension to PL/I for structured concurrent programming also based on the MONITOR concept.

*Concurrent AND/OR Programs (CAOP) [Harel and Nehab, 82].

A functional concurrent specification/programming language which utilizes recursive functions. CAOP can be considered as non-procedural. However, communication is still expressed in terms of individual messages. It resembles concurrent PROLOG in many aspects. The computation model of this language adopts the basic concepts suggested by Milner for CCS.

*ADA programming language [Ledgard, 80],[Taylor, 83]

A general purpose programming language for computer embedded systems which typically require concurrency and real time operation [Ledgard, 80]. The primary interprocess interaction is termed "rendezvous". A "rendezvous" is a match of named entries called by one task and declared in another task. A "rendezvous" is completed when a process executes an ACCEPT statement in the callee. The major concurrent computation description in ADA is through TASK description. According to [Hilfinger, 82], this is an unnecessary complication to the language and may be well defined by the existing TYPE mechanism in ADA. Structured programming technique is encouraged by the language design. Also efforts to verify the correctness of ADA programs have been made [Taylor, 83].

*Specifying Concurrent Program Modules [Lamport, 83]

A specification method intended to specify the properties of concurrent systems(safety properties and liveness properties) using intuitive temporal logic notions and power domain construction. There is some similarity between this method and the one proposed here since both approaches require making assertions about the data rather than describing the behavior of the processes.

*Calculus of Communicating Systems (CCS) [Milner, 80].

A calculus of describing mathematical models for processes and observable behavior of a concurrent system based on the notion of "flow algebra" [Milner, 79]. The objective was that the proof techniques for reasoning concurrent sequential processes could be fully developed in CCS. The concept "behavior observation" in CCS has been adopted in PFL, CAOP, and other systems.

*PAISLey, Executable requirements for embedded systems [Zave, 1982]

The result of the execution of a PAISLey specification is a set logical consequences derived from the specification. It is a language to record the requirements in a system (including supporting system and application system) in a formal way. It is not intended to be a "design specification" language, namely is not intended to really implement any actual algorithms.

*PFL: A Functional Language for Parallel Programming [Holmstrom,S. 83]

A parallel functional programming language with the intention of formal description of concurrent programs. It is built on the top of an existing functional programming language ML. It adopts some concepts from CCS, such as "channel" and "behavior". The new extension of the concurrent part inherits the rigour of ML system. It uses "typed channels" and models the imperative part of the language very carefully (the I/O part) by using continuation in denotational semantics. It is claimed by the author that PFL is more general than CCS but more difficult to reason about formally and informally.

*ON THE RELATIONSHIP OF CCS AND CSP [Stephen D. Brookes, 1983]

This article addresses the relationship between the failure model proposed for CSP and the synchronization tree model for CCS. It finds a suitable set of axioms for the failure

equivalence relation (similar to Milner's observation equivalence). This work reveals the similarity in the underlying semantic models of CCS and CSP.

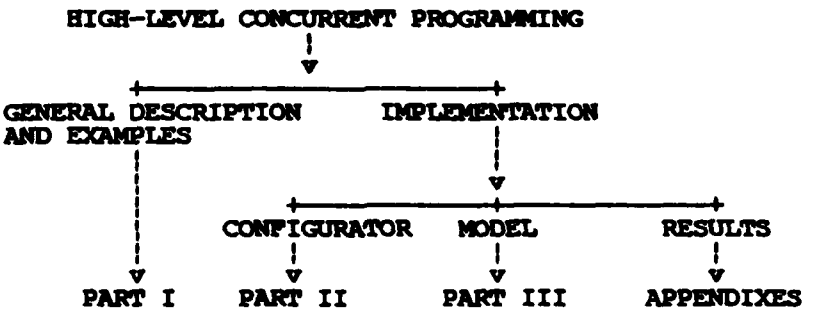
*ARGUS: THE PROGRAMMING LANGUAGE AND SYSTEM [Liskov, 83,84]

A programming language and compiler designed to solve failure and recovery problems in distributed computing. The two mechanisms, GUARDIAN and ACTION (two abstract datatypes) are used for implementing surviving services for system failure. The approach is based on ATOMICY of program units.

More thorough surveys of models and programming languages for concurrent computation can be found in [D.B. MacQueen, 1979] and [G.R. Andrews, 1983].

1.8 ORGANIZATION OF THE DISSERTATION

The dissertation is organized in the following way.



In Part I, the style of the high-level concurrent programming is presented by giving two characteristic examples of concurrent programming. Also overall description of the design of the two developed systems and the major problems solved during the development are all included.

The material in Parts II and III presents the methods, algorithms

and techniques used in the implementation of the Configurator and MODEL compiler, respectively. The algorithms used in implementation are given with estimated complexity. Reading of Part II and III is not necessary if only understanding of the general ideas is desired.

In order to let the interested reader examine the working environment of the two systems in even greater detail, actual input/output of the two systems are provided in the appendixes. Also, for the sake of completeness, the syntax descriptions of the two languages (CSL and MODEL) are given in Part II and Appendix B respectively.

CHAPTER 2

COMPOSING A CONFIGURATION OF MODULES AND FILES

2.1 MODULES AND FILES

A user composes a concurrent system by specifying a configuration of modules and files. The optimal partitioning of an overall specification into concurrent modules, to obtain low computation time, is still an open problem. Therefore typically, boundaries of modules are defined along functional divisions. The user considers each module independently in isolation. Therefore he regards the outside environment purely as data files. Such files can connect modules in the overall configuration. Subsystems are subconfiguration defined separately. In our example of resource allocation, the five philosophers and the resource allocator form respective modules naturally. Modules are consumers or producers of their source or target files, respectively.

2.2 CONFIGURATION OF THE DINING PHILOSOPHER EXAMPLE

The configuration for the Dining Philosophers is shown in Figure 2. Each philosopher module (P1 to P5) produces a file of requests and releases of resources (REQ_REL) and consumes a file of allocations of resources (ALLOC1 to ALLOC5). The resource allocator (R) has a target file of allocations (ALLOC) and a source file of requests and releases of resources (REQ_REL). A target/source or consumer/producer relationship is represented by a directed edge in the network. When the same file is produced by one module and consumed by another module

then the two modules become connected via the file.

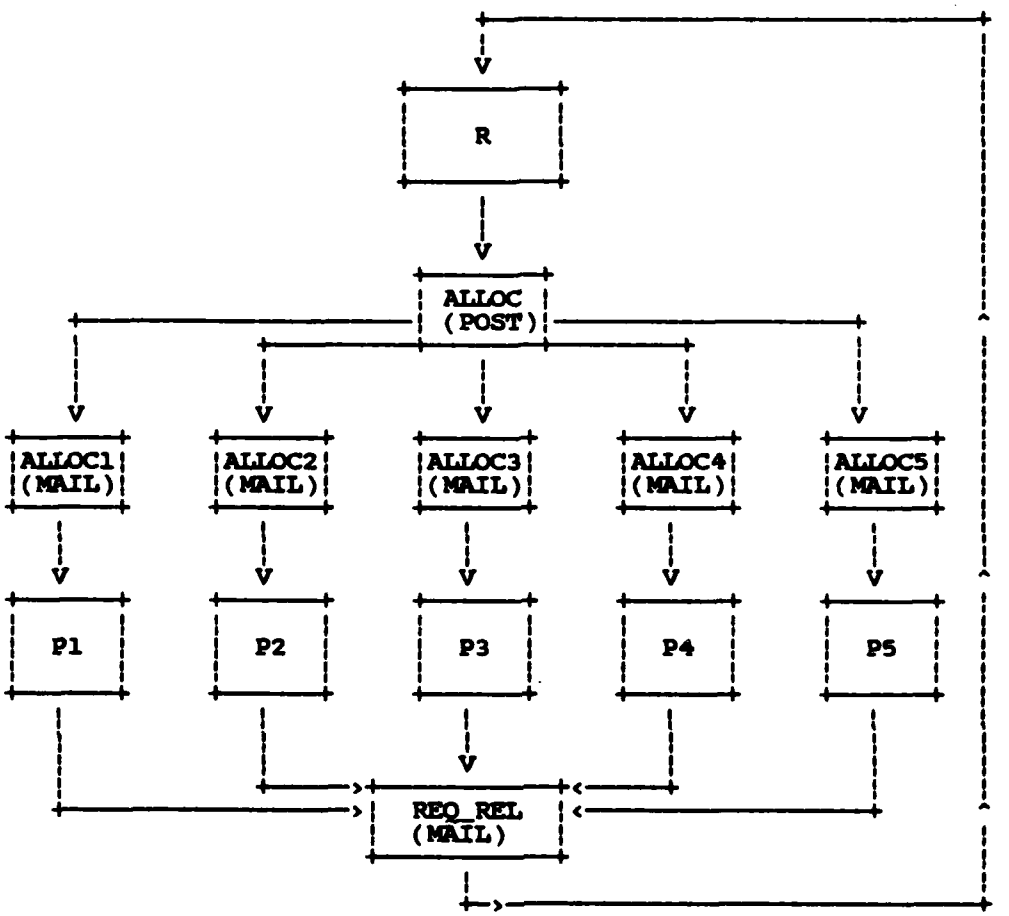


FIGURE 2. Configuration Network For Reusable Resources Allocation Example

The user must select the attributes of connecting files and in this way provide information for guiding the Configurator in attaining high degree of concurrency in the computation. The descriptions of the organization of a file, given in the specifications of it's producer and consumer modules, must be compatible. For example, as will be shown later, the target file of the resource allocator (ALLOC) and the source files of the philosophers (ALLOC1,...) contain the same data but are viewed by their producer and consumer modules as having different organizations. The file compatibility rules are are stated later as some knowledge of the MODEL language (described in Chapter 5)

is necessary.

Thus the user must instruct the system about the nature of module and file nodes. A module may be

i) an individually specified module (MDL-default),
ii) a group of modules and files that form a subsystem which is defined in a separate configuration (GRP), or
iii) a human with an interactive terminal communicating with the system (MAN). This type of "module" naturally is not initiated automatically.

As noted, the user regards files as aggregates of static data and must therefore specify the organization of the data as follows.

i) Sequential (SAM-default): The sequential file is being communicated as one entity. It implies that the file can be consumed only after it has been entirely produced. Such a file may have only one producer module, but any number of consumers. It is typically associated with a device, such as tape, printer, etc.
ii) Index-sequential (ISAM): Each record in an index-sequential file has a variable defined as a key which defines (accesses) a record in the file. There are no restrictions on the order of references to such a file by producer or consumer modules. If only a single record can be updated at a time, then the MODEL Compiler incorporates code in the generated programs to lock each other when updating the critical data. Otherwise, the user is notified and control of access must be part of the system specification (similar to the resource allocator example). The user can also indicate that an ISAM file is first produced and later consumed. In such a case the user has to define separate old and new versions of the file and denote an edge between the two versions in the configuration. An ISAM oganization file is typically associated with a disk device.

iii) A mailbox (MAIL): A mailbox file can have a number of producer and consumer modules. Records from different producers are queued in a mail file until consumed in order of their arrival. If there are more than one consumer, they consume the queued records in an arbitrary order. Thus it is not necessary to have a physical device for storing a mail organized file.

iv) Post office like facility (POST): A post file is a distributor of records to other (source) mailbox files. It has one producer module, and its records include a variable defined as an address of a destination MAIL file. Therefore, it can have any number of edges connecting it to mail files.

MAIL and/or POST file organizations are used for direct connection of files between modules without the use of intermediate storage device. The producing and consuming may then be concurrent.

The POST and MAIL files use limited space in main memory. The MODEL Compiler, when generating a program for a module, optimizes the use of main memory space used for data in these files. Program optimization causes a producer module to store and produce one or a few records at a time and the consumer module to consume and store one or a few records at a time (if possible). If producer and consumer processes are concurrent, the POST or MAIL facilities need to buffer only a limited number of records. This is similar to the concept of a pipeline or a stream. Such a file is referred to in the following as having a virtual dimension along which only a window of records needs to be buffered. The user is not involved in program design, but is told that to attain better efficiency only certain forms of subscript expressions may be used in referencing variables in such files. A specifier of a module is advised by warning messages if other subscript expression forms were used and whether a file dimension can or can not be virtual. This is further discussed later.

The above rules must be followed in connecting modules and files into a configuration. A configuration network is shown in Figure 2 for the resource allocation example. The language used to specify a

configuration consists of statements that define paths in the network. Figure 3 shows the specification for the configuration of Figure 2. A statement of the language consists of node names - prefixed by 'M:' and 'F:' to indicate whether the node is a module or a file, respectively, and suffixed by desired attributes - connected by edges '->'. The statement terminates with a ';'.

```
1   CONFIGURATION: REUSABLE;

2   M:+P1,+P2,+P3,+P4,+P5
      ->F:REQ_REL(ORG:MAIL)
      ->M:R->F:ALLOC(ORG:POST)
      ->F:ALLOC1(ORG:MAIL),ALLOC2(ORG:MAIL),ALLOC3(ORG:MAIL),
          ALLOC4(ORG:MAIL),ALLOC5(ORG:MAIL);

3   F:ALLOC1->M:P1;

4   F:ALLOC2->M:P2;

5   F:ALLOC3->M:P3;

6   F:ALLOC4->M:P4;

7   F:ALLOC5->M:P5;
```

FIGURE 3. Specification of Configuration of Figure 2

A node in the configuration graph may have a number of optional attributes, especially a physical name providing location, device, directory, version and record size (described in Part II). Default values are assumed if these attributes are not provided (which is the case in Figure 3). Also synonymous names may be declared. Module node names may be preceded by the + sign to indicate that the module is not to be initiated automatically by the command language programs produced by the Configurator, but instead will be initiated manually. In such a case the manually initiated module must give its identity in the connecting file(s). Thus the absence of such a module would not effect other modules.

For example, a philosopher (P1 to P5) module need not be initiated automatically with the resource allocator module (R). It may be initiated when the Philosopher joins the dining arrangement,

and terminated when he decides not to eat there again.

CHAPTER 3

OPERATION OF THE CONFIGURATOR

3.1 FUNCTIONS AND PHASES OF THE CONFIGURATOR

This chapter provides an overview of the Configurator. Part II gives more detail and systematic description.

The Configurator has five functions: checking the input configuration, scheduling execution of modules, evaluating diameters of strongly connected components (to be used in the iterative solution to the distributed simultaneous equations), generating JCL and PL/I programs and generating user system documentation.

The first phase of the Configurator performs syntactic checking of individual statements and constructs a configuration graph where the nodes are assigned all the necessary attributes, supplied in the specification or determined by default (Section 11.4).

The second phase analyses the graph and verifies that the rules for composing a complete and consistent specification of a configuration (Section 10.6).

Deeper global checking is conducted as follows. Maximally strongly connected components (MSCC) in the configuration graph are identified and the user is warned that they constitute a necessary but not sufficient condition for a deadlock (a deeper check is conducted by the MODEL Compiler for each of the modules in the MSCC, described later). Warning and error messages are couched in the configuration language and do not refer to implementation level concepts (Section

11.6).

In the third phase, the Configurator schedules the entire system. It attempts to minimize the usage of mailbox space. This is based on the limited information available in the configuration, although better efficiency could be obtained if their intercommunication pattern was known in detail. Processes of modules connected by post or mail files are initiated together and operate in parallel if possible. Such module nodes form a parallel component and are represented by a single node in a component graph. Modules prefixed by + sign are initiated manually. If an edge exists between the two ISAM (standing for Indexed Sequential Access Mechanism) files, it implies that completion of the producer modules must precede initiation of the consumer modules. This graph consists of nodes, each representing a module or a group of modules in a parallel component, and edges indicating sequential order between nodes. This component graph is checked for cycles, and error messages are issued if any cycles are found (Section 11.7).

The Configurator then calculates diameter for each strongly connected components in a configuration. The diameters are needed in the distributed termination algorithm (Section 11.8).

In the next phase, command language statements are generated to run the entire configuration of programs and files in a chosen environment. The program generation phase uses the available facilities offered by the operating systems and communications software as well as the available processors and communication links. In the case of the implementation using VAX/VMS, both, sharing memory or sending and receiving messages, are available for communications between modules. The technique of code generation can conveniently express either implementation strategy. The message communication method was selected as it is more suitable for a geographically distributed network and it retains better independence of a program from the types of devices used; for instance, a MAIL file may serve as a sequential file (without user intervention), depending upon

whether it is used to connect modules in a configuration or not (Section 11.9).

Due to the particular facilities in VAX/VMS, the programs generated by the Configurator consist of:

i) PL/1 programs to establish the necessary mailboxes.
ii) command language programs which initiate and synchronize sequential module or subsystem execution.

The command language programs are placed in respective files. These files form a tree structure, where each file at a non-terminal node executes the files in the nodes below it. Thus the root file of the tree is the "main" command program which contains commands for executing the files which initiate subsystems or modules, and so on. However, the command program files for modules to be initiated manually are not present in the tree. They are referred by the user for execution. In addition to the command language programs generated by the Configurator, there are PL/I program files for each module generated by the MODEL Compiler.

A module reading a record from a mailbox is suspended if the mailbox is empty, until a record has been written by another process to the respective mailbox. A module is suspended when writing a record to a full mailbox, until a record in the mailbox has been read by another process and space has become available. The latter suspension is not necessary if the space in the mailbox is unlimited.

The above communication protocols synchronize the concurrent processes. Sequential order of execution is obtained by using the synchronization facility in VAX/VMS command language [VAX/VMS, 80]. It assures that a predecessor process is completed before a successor process is initiated.

Finally a number of reports document the configuration specification, its network, the modules, the files and their

attributes, the schedule and the compatibility requirements imposed on files that connect modules.

The remainder of this chapter describes the main problems encountered in the design and implementation of the Configurator and the methods used to resolve these problems.

3.2 CHECKING

The checks performed by the Configurator are divided into the following classes:

i) Completeness and inter-module connections
ii) Consistency of (derived) temporal relations
iii) Compatibility of interfacing file descriptions

3.2.1 COMPLETENESS AND INTER-MODULE CONNECTIONS

The completeness check detects the existence of isolated nodes in a configuration (Section 11.5). The Configurator also checks connection patterns among modules and connection restrictions for each node. Basically, the following consume/produce patterns are allowed for the different file types:

MAIL	n:1
POST	1:n
SAM	1:n
ISAM	n:m

Similar restrictions have been made for MODULE nodes and the summary of the restrictions can be found in Section 10.5.

3.2.2 CONSISTENCY OF TEMPORAL RELATIONS

The underlying assumption used in the consistency checking is that all the modules in a configuration are ATOMIC (section 11.6.1),

namely they acquire all their input files on initiation and release them on termination. There are also three temporal relations defined on five basic module-file connections patterns (section 11.6.1). Let a pair of real numbers <Mis,Mie> be the starting and ending times of module Mi, the three temporal relations and implied execution time constraints are:

i) sequential relation, denoted as Mi => Mj,
   implies Mie < Mjs.

ii) mail relation, denoted as Mi -> Mj,
   implies Mjs <= Mis & Mje >= Mie.

iii) parallel relation, denoted as Mi || Mj,
   implies Mis=Mjs and Mie=Mje.

The transitivity of these temporal relations are defined (section 11.6.1.2). The temporal relations are propagated in a configuration graph according to those transitivity rules.

An inconsistency in a configuration graph is obtained by deriving either Mi=>Mi or Mi->Mj and Mi=>Mj based on the transitivity rules (Section 11.6).

3.2.3 COMPATIBILITY OF INTERFACING FILE DESCRIPTIONS

Because of the independent development of individual modules, the checking of the compatibility of interfacing file descriptions is rather difficult. However, messages are issued to warn the user of this requirement. Documentation is produced to show for each file, it's consumer and producer where compatibility of file structure is required (Section 10.6).

3.3 OPTIMIZATION

The Configurator uses the component graph to schedule module. Processor and memory usage is optimized by calling a module as late as possible when its output is needed. The concurrency of the overall system is also optimized by the use of the component graph (Section

11.6.7).

3.4 DIAMETER EVALUATION

The diameter of each strongly connected component in the configuration graph is needed for the distributed termination of the iterative multi-node solutions (Section 16.2). The Configurator calculates the diameters of the strongly connected components in a configuration and passes the diameters to the generated JCL programs, to be used by individual modules at runtime (section 11.8).

3.5 CODE GENERATION

In this phase, the Configurator generates JCL and PL/I programs for the execution of a user system. The "tree" structured execution pattern is accomplished by use of the commands in VAX/VMS. Detail description of the code generation part of the Configurator is given in section 11.9 (Part II). Example JCL and PL/I programs generated from a CSL specification are given in Appendix A.

CHAPTER 4

SPECIFYING INDIVIDUAL MODULES - RESOURCE ALLOCATOR

To complete implementation of the configuration of Figure 2, it is necessary to specify each module independently. In this configuration there is a philosopher module, which repeats five times, one for each of the five philosophers, and a resource allocator module. The specifications of these modules are discussed below. This chapter provides an introduction to the use of the MODEL language.

4.1 THE PHILOSOPHER MODULE

Figure 4 shows the specification of the philosopher module stated in the MODEL language. The specification is divided for convenience into five parts: header, data description, data parameters and internal and external equations. There are also explanatory comments and statement numbers in Figure 4. The rational behind composing the statements is discussed in the following.

The header consists of the name of the module (Pk), the source file of allocations (ALLOCk) and target file of requests and releases (REQ_REL). The lower case k denotes the unique number of each philosopher.

```
    /*HEADER*/

1     MODULE: Pk(k=1 to 5);  /* module name (repeats 5 times)  */
2     SOURCE FILE: ALLOCk;   /* allocation files               */
3     TARGET FILE: REQ_REL;  /* file of requests and releases  */

    /*DATA DESCRIPTION*/

4     1 ALLOCk IS FILE(ORGANIZATION IS MAIL)
        2 MSGA(*) IS RECORD, /* individual allocation message  */
          3 PROC_ID IS FIELD(PIC'9'), /* process identifier    */
          3 CLOCKA IS FIELD(BIN FIXED); /* time of allocation  */
5     1 REQ_REL IS FILE(ORGANIZATION IS MAIL)
        2 MSGR(*) IS RECORD, /* request/release message        */
          3 PROC_ID IS FIELD(PIC'9'), /* req/rel process id    */
          3 RQ_OR_RL IS FIELD(BIT(1)), /* request=0,release=1  */
          3 RES(5) IS FIELD(PIC'9'), /* quantities of resources*/
          3 CLOCKR IS FIELD(BIN FIXED); /* req/rel time        */

    /*DATA PARAMETERS*/

6     (I,J) ARE SUBSCRIPTS;      /* I for MSGR,   J for RES     */
7     IX IS FIELD(FIXED BINARY) /* indirect(sublinear)subscript*/
8     IX(I)=IF I=1 THEN 1
          ELSE IF RQ_OR_RL(I-1) THEN IX(I-1)+1
          ELSE IX(I-1);          /* index of MSGA               */
9     END.MSGR(I)=RQ_OR_RL(I) & RANDOM>.99;
                          /* definition of the range of MSGR */

    /*EQUATIONS FOR VARIABLES IN FILE REQ_REL*/

10    PROC_ID(I)=<k>;
11    RQ_OR_RL(I)=IF I=1 THEN '0'B ELSE ^RQ_OR_RL(I-1);
12    RES(I,J)=(J=MOD(k,5))|(J=MOD(k+1,5));
                          /* right and left fork request */
13    CLOCKR(I)=IF I=1 THEN TIME
          ELSE IF RQ_OR_RL(I) THEN CLOCKA(IX(I-1))-1000*LOG(RANDOM)
          ELSE CLOCKR(I-1)-10000*LOG(RANDOM)


     /*EQUATION DEFINED EXTERNALLY (IN OTHER MODULES)*/

14    MSGA(IX(I))=DEPENDS_ON(MSGR(I));
```

FIGURE 4. Specification of Philosopher Module

The data description part in Figure 4 declares the structure of the two files. A data structure is described hierarchically as a tree. The apex node is called FILE, an intermediate node is either a GROUP or a RECORD. A RECORD is the smallest structure exchanged between an external environment or device and the module. A terminal node is denoted as a FIELD. Each of the nodes is named, and may repeat and form a vector. The number of repetitions, or size of the vector is in parenthesis following the name. "*" indicates an unknown (variable) number of repetitions. The primitive data types are

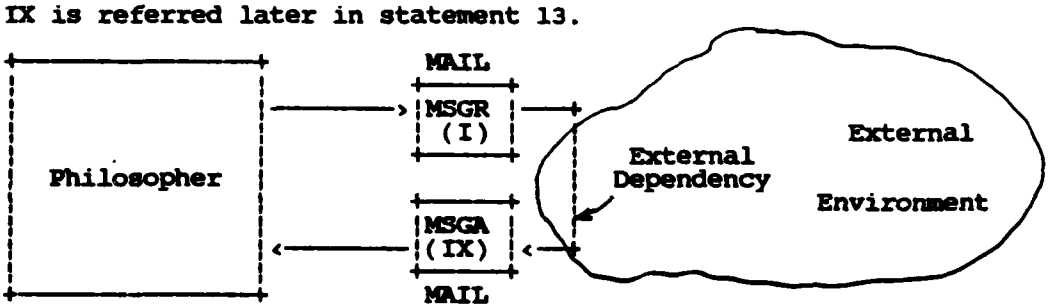
similar to PL/I - picture, decimal (fi loat), binary, bit and character. Thus the ALLOCk file (stateme ontains a vector of allocation messages (records) MSGA. The EQ_REL file (statement 5) contains a vector of requests/releases messages (records) MSGR. These two entire vectors are viewed by the user as they were available a priori and his main task is to compose equations which relate them.

A philosopher requesting/releasing resources/forks identifies himself in the PROC_ID field of MSGR. The philosopher to whom resources are allocated (i.e. by the resource allocator module) is identified in the PROC_ID field of MSGA. The records in the ALLOCk file come from a post organized file. MSGR may be for a request or a release of resources, and RQ_OR_RL denotes which case it is. Each MSGR includes a vector of resources RES, which contains the quantity of each resource that is requested or released. There are 5 resources in the problem of the 5 dining philosophers - each consisting of a single fork in a respective position. Finally, CLOCKA and CLOCKR are used to simulate the clock (in seconds) of an allocation and request/release, respectively.

Repeating data structures form arrays. The individual elements of these arrays are referred to by use of subscripts. The sizes of dimensions of arrays may be variable and need to be defined. They constitute the data parameters of the specification in Figure 4. Statement 6 declares two free variables I and J that are used as subscripts. They assume all the integer values from 1 to the size of the dimension of the variable which they index. Note that they differ from ordinary variables which can assume only a single value. I is used to subscript the request/release messages, (MSGR and its constituents), and J to subscript the resources, RES. Note that RES, the requested or released resources, changes for each message and therefore is two-dimensional, with subscripts I,J. I indexes the "historical" values of RES. There is a correspondence between individual allocations and requests/releases. For each requesting MSGR, (where RQ_OR_RL=0), there is a corresponding allocation MSGA.

No MSGA is necessary for a release MSGR, (with RQ_OR_RL=1).

A widely used method in MODEL for relating elements in two arrays is to define separately the indices of the related elements. This is the case in defining an indirect index vector IX (an internal variable) which gives the indices of MSGA for each index I of MSGR. IX is declared in statement 7 and defined in statement 8. IX is a vector of the same shape as MSGR. Thus it has a value for each value of I. For I=1 it has a value 1. Then, IX is increased by one if the preceeding MSGR is a release. We call IX sublinear to I. The sublinear relation between IX and I satisfies two conditions: IX(1)=0|1 and IX(I)-IX(I-1)=0|1. The program generator recognizes sublinearity and uses it to generate a more efficient object program. IX is referred later in statement 13.



a) Illustration of External Dependency Edge

Subscripts		Record		Record
I	IX(I)	MSGR	Dependency	MSGA
1	1	request 1	external(R)	allocation 1
2	1	release 1	internal(P)	
3	2	request 2	external(R)	allocation 2
4	2	release 2	internal(P)	
.	.	.	.	.
.	.	.	.	.
.	.	.	.	.

b) Indices of Records in the External Dependency Statements

FIGURE 5. External Dependency As Viewed From The Philosopher Module

Figure 6(b) illustrates the relations of these subscripts and records. (The Dependency column is described later).

A condition of the last element in a dimension is denoted in MODEL by a variable named by prefixing END to the name of a variable with the rightmost (lowest order) dimension. This variable has the same shape as the one named in it's suffix. All it's elements have a value 0, except for the last element in the rightmost dimension which has value 1. Statement 9 defines END.MSGR, which has the same shape as MSGR and effectively gives the size of MSGR. It expresses an assumption that a philosopher, after having dined repeatedly, on the average 100 times, exponentially distributed, has had enough and decides to quit and dine elsewhere. Thus every element in END.MSGR(I) has a value of 0, except the last element which has a value of 1.

Statements 10 through 13 define the four FIELD variables in the REQ_REL file. PROC_ID is the philosopher identification. The value of RQ_OR_RL is 0 for a request and 1 for a release. The requested forks in RES are always to the left and right of the philosopher, as expressed in statement 12. CLOCKR simulates the time stamp of a request or of a release of resources. Statement 13 shows that for I=1, CLOCKR is the time of the first dining request (defined by the function TIME), otherwise it depends on the time of the previous allocation (CLOCKA(IX(I-1))) and the dining and thinking times which are assumed to be exponentially distributed with 1000 and 10000 seconds means respectively. Note that this assumes that a philosopher may join and quit the diners at any time of his choice and the number of philosophers may be variable. However, each philosopher must have a seat assigned at the table in advance of the first eating.

To specify the philosopher module completely it is further necessary to specify external dependencies due to functions provided by other modules. The functions provided by the outside environment, however complex they may be, interest the module specification only to the extent of knowing that they exist. While the outside relation may change, as long as the dependency continues to hold it is not

necessary to respecify the module. In our example it is necessary to specify in a philosopher module the external function of an allocation in response to a respective request (this dependency is imposed by the resource allocator module). It is not necessary to show this relationship in detail as expressed in the R module. A user can express it in a reduced form as shown in statement 14 of Figure 4. The pseudo function DEPENDS_ON is used to express the fact that the source record variable(s) on the left hand side depend externally on the target record parameters of the function. Note that the internal dependency of a release on an allocation is expressed implicitly in equation 13. These two dependencies are illustrated by the labeled arrows in the table at the bottom of Figure 5.

4.2 THE RESOURCE ALLOCATOR MODULE R.

Figure 6 shows the specification of the resource allocator module R. The R module is larger and more complex than the philosopher module. It further illustrates the equational style.

Statements 1-3 in Figure 6(a) give the name of the module, R, the source file REQ_REL, and target file ALLOC. Another target file SIMULATION is a report of the results of the simulation of the Dining Philosophers problem. The specification of SIMULATION is given in statements in Figure 6(c). REQ_REL and ALLOC are declared in statements 4 and 5 of Figure 6(a). REL_REQ consists of the combined requests/releases received in the mail from all other modules in the sequence of their arrival. ALLOC consists of all the allocations of resources distributed through the post office like facility to all the modules in the order that they are issued. Note that the records in ALLOC form a two dimensional ragged edge matrix, with each row corresponding to all the allocations that can be made in response to a respective request or release message. This differs from the vector organization of ALLOCk. However this does not violate the rules of compatibility of communicating files.

```
   /*HEADER*/

1  MODULE:R;
2  SOURCE:REQ_REL;    /* merged requests/releases from all processes */
3  TARGET:ALLOC,      /* merged allocations to all processes         */
          SIMULATION;/* report of results of simulation             */

   /*DATA DESCRIPTION*/
4  1 REQ_REL IS FILE, (ORG IS MAIL),
    2 MSGR(*) IS RECORD, /* messages for req/rel of resources */
      3 PROC_ID IS FIELD (PIC'9'),    /* id of process       */
      3 RQ_OR_RL IS FIELD (BIT(1)),   /* request=0,release=1 */
      3 RES(5) IS FIELD (PIC'9'),     /* vector of resources */
      3 CLOCKR IS FIELD (PIC'(9)9'); /* time of message     */
5  1 ALLOC IS FILE, (ORG IS POST, KEY IS PROC_ID),
    2 MSGAS(*) IS GROUP,              /* group of alloc messages */
      3 MSGA(*) IS RECORD,            /* individual message  */
        4 PROC_ID IS FIELD (PIC'9'),  /* allocated process   */
        4 CLOCKA IS FIELD (PIC'(9)9');/* time of allocation  */
6  1 QUEUE IS FILE,                   /* process queues      */
    2 STAT_Q(*) IS GROUP,             /* queue for each req/rel */
      3 PROC(*) IS GROUP,             /* process in queue    */
        4 PROC_ID IS FIELD (PIC'9'),  /* id of process       */
        4 IN_IX IS FIELD (PIC'9'),/*index of process in queue */
        4 OUT_IX IS FIELD (PIC'9'),   /* index of process in alloc */
        4 RES(5) IS GROUP,            /* resource vector     */
          5 CLAIM IS FIELD (PIC'9'),  /* maximum resources claimed */
          5 SUM_CLAIM IS FIELD (PIC'9'),
                          /* sums of claims for resources in q */
        5 SAT IS FIELD (BIT(1));      /* availability of resources */
7  1 RES_LIMIT IS GROUP,
    2 NUM_RES(5) IS FIELD (PIC'9'); /* # of resources available */


    /*DATA PARAMETERS*/

8  (I,J,K,L) ARE SUBSCRIPTS;
                     /* I subscript of request/release messages */
                     /* J subscript of resources                */
                     /* K subscript of processes in queue       */
                     /* L subscript of group of allocations     */
9  SIZE.PROC(I)=IF I=1 THEN 1
                   ELSE IF RQ_OR_RL(I) THEN SIZE.PROC(I-1)-1
                                       ELSE SIZE.PROC(I-1)+1;
                     /*        size of process queue          */
10 SIZE.MSGA(I)= IF SIZE.PROC(I)>0 THEN  OUT_IX(I,SIZE.PROC(I))
                   ELSE 0;
                     /*        size of group of allocations   */
11 NUM_RES(J)=1;     /*        one fork in each position      */
```

FIGURE 6(a) Resource Allocator Module Specification: Header, Data Description and Definition of Data Parameters

The file compatibility rules are briefly summarized below.

i) The data structures that constitute the unit of transfer of information between different media in a computer system are denoted as records. A match must be possible between the variables in the corresponding records in producing and consuming

module specifications. The lengths (in bytes) of matching records, specified separately in the consumer and producer modules, must be the same.

ii) Matched variables in the respective records may be named differently in the producer and consumer module specifications, but they must have the same attributes, i.e. data type, scale and length.

iii) Matched records may form arrays in respective specifications. It is not necessary that the number of dimensions of the arrays in the specifications of the file in consumer and producer modules be the same, but the total number of records must be the same.

There is also an internal QUEUE file consisting of the history of the status of the queue of processes. It repeats for each request/release. Processes are added and retained in the queue in the order of the respective requests, and omitted as a result of respective releases. The QUEUE file is described in statement 6. STAT_Q is the status of the queue for each request/release message. The individual entry in the queue is PROC. It contains the identification of the process PROC_ID. Two indirect index variables, IN_IX and OUT_IX are described further below. A vector RES contains information on requested resources. RES is a matrix with rows corresponding to processes and columns corresponding to resources. The components of RES, i.e. CLAIM, SUM_CLAIM and SAT, are therefore also matrices. (Actually 3 dimensional, repeated for each request/release). CLAIM is the number of resources claimed by the process. SUM_CLAIM is the cumulative number of resources needed to satisfy all the claims by this process and it's predecessors in the queue. SAT is a binary variable indicating for each process whether the claims for a resource and it's predecessors resources, in the order of resources in RES, can be satisfied from available resources.

It is typical in MODEL to specify permutation or selection of elements of a vector by defining the indices of the respective

elements. IN_IX is the index of a process in the preceding queue, i.e. in STAT_Q(I-1). A number of processes may be allocated resources as a result of a release message. OUT_IX is the index of the process in the respective group of allocation messages. Both, IN_IX and OUT_IX increase monotonically with the order of the processes in the queue. These variables are discussed further.

A number of parameters are used with, or are attributes of, the data. The subscripts I,J,K and L are declared in statement 8. The subscript I indexes request/release messages. J indexes resources (forks). K indexes positions of processes in the queue and L indexes allocation messages. There are three dimensions that require definition of their sizes. The range of I is assumed as infinity reflecting the notion that R will operate forever. There are a number of ways to define a size of a dimension in MODEL. The use of the END prefixed variable was already presented in Figure 4. Another way to define a size is through prefixing the keyword SIZE to the name of a respective data structure. SIZE prefixed variables define the number of elements in the respective dimension. The size of the vector PROC, i.e. the number of processes in a queue, is defined in statement 9. As shown in statement 9, the size of the queue increases by one for each request and decreases by one for each release. The size of the vector MSGA, i.e. the number of allocations in a group is defined in statement 10. The size of the group of allocation messages, MSGA, is the same as the value of OUT_IX for the last process in the queue. This is discussed further below. Figure 6(a) ends with definition of RES the number of resources of each type - namely there is 1 fork for each of the five fork positions.

Figure 6(b) shows the equations for variables in the two files QUEUE and ALLOC and the external dependencies. There are several same named variables in different files and the name of the respective file is used as a prefix to remove the ambiguity. QUEUE.PROC_ID, the identity of a process in the queue, is defined in statement 12 for two cases - for adding a process corresponding to a request, or for

retaining a process in the queue. IN_IX gives the index of the process in the preceeding (I-1)th status of the queue.

```
   /* EQUATIONS FOR VARIABLES IN FILE QUEUE */

12 QUEUE.PROC_ID(I,K)= IF ^RQ_OR_RL(I) & (K=SIZE.PROC(I))
                       THEN REQ_REL.PROC_ID(I)
                       ELSE QUEUE.PROC_ID(I-1,IN_IX(I,K));
13 IN_IX(I,K)= IF ^RQ_OR_RL(I) THEN K
               ELSE IF REQ_REL.PROC_ID(I)^=QUEUE.PROC_ID(I-1,K)
                    THEN IF K=1 THEN 1 ELSE IN_IX(I,K-1)+1
                    ELSE IF K=1 THEN 2 ELSE IN_IX(I,K-1)+2;
14 OUT_IX(I,K)=IF RQ_OR_RL(I) THEN
                  IF SAT(I,K,5)& ^SAT(I-1,IN_IX(I,K),5)
                       THEN IF K=1 THEN 1 ELSE OUT_IX(I,K-1)+1
                       ELSE IF K=1 THEN 0 ELSE OUT_IX(I,K-1)
               ELSE IF K=SIZE.PROC(I)&SAT(I,K,5) THEN 1 ELSE 0;
15 CLAIM(I,K,J)= IF ^RQ_OR_RL(I) & (K=SIZE.PROC(I))
                   THEN REQ_REL.RES(I,J)
                   ELSE CLAIM(I-1,IN_IX(I,K),J);
16 SUM_CLAIM(I,K,J)= IF K=1 THEN QUEUE.CLAIM(I,K,J)
                     ELSE QUEUE.CLAIM(I,K,J)+SUM_CLAIM(I,K-1,J);
17 SAT(I,K,J)=IF J=1
    THEN(SUM_CLAIM(I,K,J)<=NUM_RES(J)|QUEUE.CLAIM(I,K,J)=0)
    ELSE SAT(I,K,J-1) &
        (SUM_CLAIM(I,K,J)<=NUM_RES(J)|QUEUE.CLAIM(I,K,J)=0);

   /* EQUATIONS FOR VARIABLES IN FILE ALLOC */

18 ALLOC.PROC_ID(I,OUT_IX(I,K))=
       IF (K=1 & OUT_IX(I,K)=1)|(K>1&OUT_IX(I,K)>OUT_IX(I,K-1))
       THEN QUEUE.PROC_ID(I,K);
19 CLOCKA(I,L)=CLOCKR(I);

   /* EQUATIONS FOR EXTERNAL DEPENDENCIES */

20 MSGR(I)=DEP_ON(MSGA(I-1,L));
```

FIGURE 6(b) Resource Allocator Module Specification: Equations For Variables in Files QUEUE and ALLOC and the external dependencies

When the Ith MSGR corresponds to a request then the requesting process is added in the last position. In the case of a release, the resource releasing process is deleted from the queue. OUT_IX, defined in line 14, is monotonically increasing along the queue. Processes in the queue that are not being allocated resources have an OUT_IX value equal to the preceding process, while OUT_IX is increased for processes which are allocated resources. As it is easy to verify, in the case of the dining philosophers, there may be 0, 1 or 2 allocations for each I.

As defined in statement 15, CLAIM, the maximum number of resources of each type requested by a process, is contained in the respective request and also retained in the queue. SUM_CLAIM, defined in statement 16, is the cumulative number of resources of each type needed to satisfy the claims of a process and all its predecessors in the queue. If SUM_CLAIM(I,K,J) exceeds NUM_RES(J), the total number of copies of a resource, then the respective CLAIM(I,K,J) can not be satisfied. This condition is used to define SAT in statement 17.

Statements 18 and 19 define the identity of the process that is allocated resources (ALLOC.PROC_ID), and the simulated time of the allocation (CLOCKA). Statement 19 neglects the computation time for computing the allocation. As noted, the latter is useful in a simulation report of the Dining Philosophers problem.

Finally the equation for the external dependency, as seen by the R module is shown in statement 20. This equation expresses the outside dependency of a release on a previous allocation. It is illustrated in Figure 7.

One of the advantages of this style of programming is the ease of making complex changes. For instance, if we wanted to give priority in allocations to some modules, we would have to add priority variables to the declaration of requests (statement 4) and only modify the equation for SAT (statement 17) to include the dependence on the priority variables.

As noted, the dependency need not express fully the allocation-release relations. In the R module, the dependency is visualized in terms of the combined request/releases file, received from all the philosopher's modules. Namely, the index of allocation to a Philosopher must be less, at least by 1 (i.e.,I-1) than the index of a release.

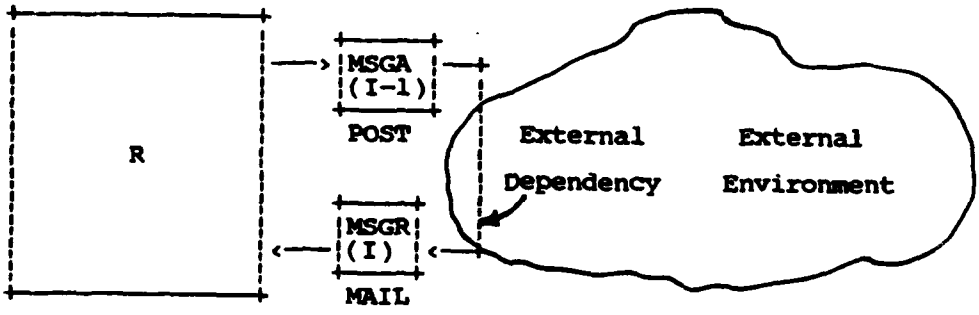
To obtain a printed output file of the simulation results of the Dining Philosophers problem, it is necessary to define a report, called SIMULATION. As shown in Figure 6(c) statement 21, this file

contains the identification of the philosophers and the respective times of requests and allocations of forks. Statements 22-26 define the values in this file as equal to respective PROC_ID, CLOCKR and CLOCKA variables. For each request there may be 0,1 or 2 allocations.

```
21  1 SIMULATION  IS FILE,
      2 HD1 IS RECORD,
        3 HDF1 IS FIELD(CHAR 125),
      2 HD2 IS RECORD,
        3 HDF2 IS FIELD(CHAR 125),
      2 HD3 IS RECORD,
        3 HDF3 IS FIELD(CHAR 125),
      2 EVENT(*) IS RECORD,
        3 REQUEST IS GRP,
          4 PROC_IDM  IS FIELD(CHAR 4),
          4 FILLER1   IS FIELD(CHAR 1),
          4 RQ_OR_RLM IS FIELD(CHAR 3),
          4 FILLER2   IS FIELD(CHAR 1),
          4 RESM(5)   IS FIELD(PIC '9'),
          4 FILLER3   IS FIELD(CHAR 1),
          4 CLOCKRM   IS FIELD(PIC 'B(12)9'),
          4 FILLER4   IS FIELD(CHAR 2),
        3 ALLOCATION(*) IS GRP,
          4 FILLER5   IS FIELD(CHAR 1),
          4 PROC_IDA  IS FIELD(CHAR 4),
          4 FILLER6   IS FIELD(CHAR 1),
          4 CLOCKA    IS FIELD(PIC 'B(12)9');

22  SIMULATION.HDF1='          REQUEST                              AL'||
                    'LOCATION';
23  SIMULATION.HDF2='p_id R/L resrc     time        p_id     time'||
                    ' p_id      time      p_id      time';
24  SIMULATION.HDF3='==== == ===== ============ ==== ========='||
                    ' ==== ============ ==== ============';
25  (FILLER1,FILLER2,FILLER3,FILLER4,FILLER5,FILLER6) =' ';
26  SIMULATION.PROC_IDM =REQREL.PROC_IDM;
27  SIMULATION.RESM     =REQREL.RESM;
28  SIMULATION.CLOCKRM  =REQREL.CLOCKRM;
29  SIMULATION.PROC_IDA =SUBSTR(ALLOC.PROC_ID,6,1);
30  SIMULATION.CLOCKA   =ALLOC.CLOCKA;
```

FIGURE 6(c). Simulation Report Specification of the Resource Allocation Module



a) Illustration of Dependency Edge

SUBSCRIPT	MSGR	DEPENDENCY	MSGA
I			
1	p3.req1	Internal(R)	p3.alloc1
2	p1.req1	Internal(R)	p1.alloc1
3	p3.rel1	External(P3)	
4	p1.rel1	External(P1)	
5	p1.req2	Internal(R)	p1.alloc1
6	p3.req2	Internal(R)	p3.alloc2
.	.	.	.
.	.	.	.

b) Dependency Relations

FIGURE 7. Illustration of External Dependencies As Viewed From the R Module

CHAPTER 5
THE OPERATION OF THE MODEL COMPILER

The implementation of the configuration of Figure 2 requires specification of each module independently, and its submission as input to the MODEL Compiler. A brief description is given here of the key problems and methods in the MODEL Compiler. The following two chapters describe specification of modules and introduce the MODEL language.

The previously developed MODEL Compiler [Prywes, 83] was extended for concurrent programming. It performs the following major tasks in translation of a specification into the procedural program. After syntax analysis, the compiler constructs a dataflow-like graph to represent the specification in a convenient form. Based on the graph, implicit information is derived and entered, checks are conducted and an optimally efficient schedule of program execution is derived. The optimized schedule is finally transformed into a procedural program in PL/I. The generated program includes also analysis of various conditions of program failure, such as data type errors, absence of expected records, etc., and recovery from such failures.

5.1 REPRESENTATION OF THE SPECIFICATION AS AN ARRAY GRAPH

The specification is represented by an array graph, where a node represents the accessing, storing or evaluation of an entire array and the edges represent dependencies among variables. The underlying

graph of elements of the array may be derived from the array graph based on the attributes of dimensionality, range, and forms of subscript expressions, which are given for each node and edge in an array graph. A data node has as attributes the ranges for its dimensions and its data type. An equation node has as attributes subscripts and ranges corresponding to the union of subscripts of the variables appearing in the equation. A node A corresponding to a m dimensional data or equation array represents the elements from A(1,1,...,1) to A(N1,N2,...,Nm) where N1...Nm are the ranges of dimensions 1 to m respectively. Similarly a directed edge represents all the instances of dependencies among the array elements of the nodes at the ends of the edge and has as attributes subscript expressions for each dimension. The edges have the subscript expression for each dimension as attributes. The dependencies imply precedence relationships in the execution of the respective implied actions. There are several types of them. For example, a hierarchical precedence refers to the need to access a source structure before its components can be accessed, or vice-versa, the need to evaluate the components before a structure is stored away. Data dependency precedence refers to the need to evaluate the independent variables of an equation before the dependent variable can be evaluated. Similarly, data parameters of a structure (such as SIZE of a dimension) must be evaluated before evaluating the respective structure.

5.2 CHECKING COMPLETENESS AND CONSISTENCY OF A SPECIFICATION

It is inevitable that the user will make mistakes in specifying a computation, and it is necessary to have a dialog that helps the user to formulate a specification and make corrections. The automatic program generation can not be completed when a specification is inconsistent or incomplete. Therefore checking of structural consistency is conducted on a global basis with special focus on

iterative and recursive relations which usually encompass many entities in a specification. The specification-wide checks are categorized into checks of completeness, non-ambiguity and consistency. Incompleteness and ambiguity are detected in constructing the array graph while special procedures check consistency. The consistency checking of the entire specification may be essentially regarded as "propagation" of data type, dimensionality and ranges, from node to node. The problems discovered are described to the user in terms of the very high level language specification, without referring to programming details. The compiler is tolerant of many kinds of omissions. Data description statements are generated for variables referred in the equations but not described by the user. Equations are generated to relate same named input and output variables. Finally, circular logic is recognized by irresolvable cycles in the array graph (discussed further below).

5.3 OPTIMIZATION OF PRODUCED PROGRAMS

In composing a specification of a computational task, the user chooses a natural and convenient representation. It is up to the MODEL Compiler to map the user's representation into an efficient procedural computer program. An overall flow of program events is produced first in a skeletal, object language independent form called a schedule. The final program generation phase translates individual entries in the schedule into statements in the object language and further optimizes the produced program.

The general approach to scheduling consists of creating first a component graph which consists of all the maximally strongly connected components (MSCC) in the array graph and the edges connecting the MSCCs. The component graph is therefore an acyclic directed graph. This graph can be topologically sorted. There is a large number of possible linear arrangements of the schedule which have varying

efficiencies. The objective is to find a near optimal schedule.

This is done as follows. The subscripts for each node in the array graph are determined. Iterations for these subscripts must bracket the respective nodes to define all the values of the elements variables in the array nodes. Each node must be enclosed within loops, which are nested if the respective equations or data arrays are of multiple dimensions. Next, attempts are made to enlarge the scope of iterations. Nodes with the same range can be merged into larger components. Merging scope of iterations may enable sharing memory locations by elements of the same or related array variables. If it is possible to retain in memory only a window of the entire dimension of a variable, then the respective dimension is called virtual, otherwise it is called physical. When there is a number of ways that components can be merged (for different dimensions) then the memory requirements of different candidate scopes of iterations serves as the criterion for selecting the optimal scope. Virtual dimensions are found by the present MODEL Compiler only where the subscript expressions used to reference variable are of the form (I-k) (I is any subscript, k is 0 or a positive integer), or when the socalled sublinear or sawtooth indirect indices are used [Lu, 81]. The use of sublinear indices is further explained in the resource allocation example.

The MODEL compiler attempts to decompose the MSCC by deleting edges which represent dependencies already assured by the order of iterations. If the MSCC is not decomposable than the user is advised of the nodes and edges of the MSCC. It is up to the user to verify that they do not represent an inconsistency, such as circular logic. The other possibility is that they constitute a set of simultaneous or recursive equations. In the latter case they are solved by incorporating in the produced program a selected iterative solution method (currently Gause-Seidel). This is discussed further in connection with the cooperative computation example.

Additional optimization is performed to reduce computation time by further merging variables which have the same values into common memory space. It is possible thus to eliminate statements that copy values from one variable to another. Transformation of remaining statements allows sometimes the elimination of entire iteration loops [Szymanski, 84].

5.4 EXTENSIONS FOR CONCURRENT PROCESSING

In order to extend the MODEL compiler for concurrent programming, four major extensions have been made.

5.4.1 EXTERNAL DEPENDENCY

The essential difference between a module to be computed sequentially and concurrently is the impact imposed by external environment. In a dataflow representation, this impact can be represented as an external data dependency. An external dependency statement (DEPENDS_ON) is designed to express such a relationship. The specifier of a concurrent module (more precisely, a candidate module for concurrent execution) must specify the modules external requirement explicitly. The MODEL compiler can then use the same process, as described previously, to verify the consistency of external data references.

The external dependency, when entered into the array graph, may cause creation of cycles. Such cycles provide a necessary but not sufficient condition for consumable resource deadlock. The MODEL compiler therefore conducts deeper analysis of initiating and termination conditions implied by such cycles. It can then determine whether the system is safe of consumable resource deadlocks. Else, it incorporates an iterative procedure to solve them, thus prevents the consumable resource deadlocks. This is described in detail in Chapter 16.

The implementation of the external dependency statement is given

in Chapter 13.

5.4.2 THE MAIL AND POST FILES

MAIL and POST are the two new file organizations extended to the MODEL compiler. The purpose of introducing the two files is to enable a module to communicate with others concurrently. Because of the high-level semantics of these two file organizations, the user is isolated from the considerations of global inter-leavings of computation events. The same static view as composing a sequential file is utilized in composing the POST and MAIL files. The semantics of the two file organizations complements the existing sequential and index sequential file organizations. Implementation detail are given in Chapter 14.

5.4.3 CONCURRENT ISAM FILE UPDATES

Sharing an ISAM file or database concurrently requires the use of record locking mechanism, although the user does not "see" such low-level detail.

Restricted by the available operating system, only one-record locking is supported. That is, every record in a database is locked when it is being updated. This extension ensures, if a MODEL specification can be scheduled into one-record locking mechanism, proper locking will be made automatically in the produced PL/I program. Otherwise a warning message is issued. The user is advised to incorporate multi-record exclusion algorithm manually. The implementation details are given in Chapter 15. An example PL/I program generated from a concurrent MODEL specification is given in Appendix A.

5.4.4 ITERATIVE SOLUTION TO DISTRIBUTED SIMULTANEOUS EQUATIONS

This extension makes the concurrent MODEL more powerful in a distributed environment. The MODEL language is based primarily on the

notion of equations. In many applications, problem is expressed as a set of simultaneous equations distributed in a number of modules. These module must then cooperate in an iterative solution process. The MODEL compiler recognizes the need to use a single or multi-module iterative solution and incorporates the communication protocols and algorithms when necessary. Detail can be found in Chapter 16.

CHAPTER 6

A SECOND EXAMPLE - COOPERATIVE COMPUTATION

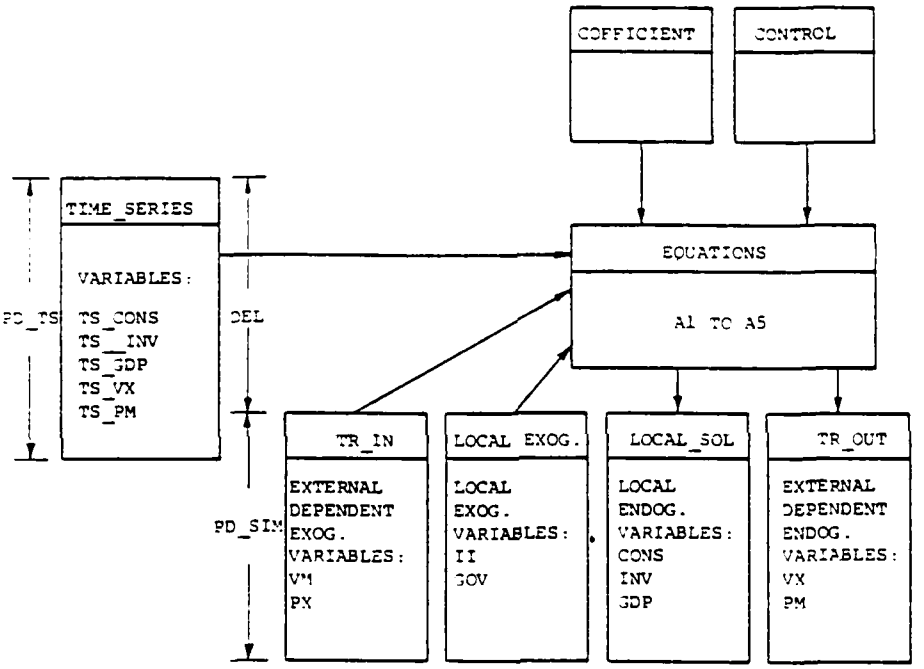
6.1 SPECIFICATION OF INDIVIDUAL ECONOMETRIC MODULES

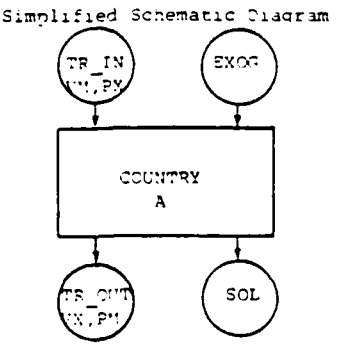
As mentioned previously, the concurrent computation in the second example represents a simulation of economic interactions in the Pacific Basin. The economies simulated and their corresponding models are those of the USA, Japan, Taiwan, Korea, Philippine and Thailand. Each model is represented in separate module. Due to space limitation we consider in Part I a reduced econometric model, shown in figure 8. The complete set of MODEL equations for each of these countries are given in Appendix A.

Although the example of econometric model consists of only five equations, it is generally characteristic of econometric models and illustrates the full models given in Appendix A. It contains also nine variables, five local and four global. The variables are listed in Figure 8. Local parameters are in the files: Coefficient, Control and Time-series. The Coefficient file contains the values of the coefficients in the equations, assumed to have been previously computed using estimation methods. The Control file contains three parameters: PD_TS, the number of periods in the Time-series file; PD_SIM, the number of periods in the simulation (forecast); and DEL, the number of periods from the beginning of the time series to the beginning of simulation. The Time-series file contains historical values for the endogeneous variables which are used as starting values for the simulation. There are two files for exogeneous variables (Local_exogen and Trade_in) and two files for endogeneous variables

(Local_solution and Trade_out). The former are prepared as source data and the latter are computed and constitute the target data. This type of a model may be represented concisely by the simplified diagram shown at the bottom-left of Figure 8.

Figure 9 shows the MODEL specification of the reduced econometric model of Figure 8.





Key:
CONS - Consumption
INV - Investments
GDP - Gross Domestic Product
II - Government Investments
GOV - Government Expenditures
VM - Volume of Imports
VX - Volume of Exports
PM - Price Index of Imports
PX - Price Index of Exports

Figure 8. A Reduced Example of a Local Country Econometric Model (Country A)

```
   /* HEADER */

1  MODULE NAME: A;
2  SOURCE FILES: TIME_SERIES, CONTROL, COEFFICIENTS, TRADE_IN,
                 LOCAL_EXOG;
3  TARGET FILES: LOCAL_SOL, TRADE_OUT;

   /* DATA DESCRIPTIONS */
4  1 CONTROL IS FILE,
    2 C_RECORD IS RECORD
     3 (PD_TS,PD_SIM,DEL) IS FIELD (PIC'999');
5  1 COEFFICIENTS IS FILE,
    2 COEF_RECORD IS RECORD
     3 C(13) IS FIELD (DEC FLOAT);
6  1 TIME_SERIES IS FILE,
    2 TS_RECORD (1:20) IS RECORD
     3 (TS_CONS,TS_INV,TS_GDP,TS_VX,TS_PM)ARE FIELDS (DEC FLOAT);
7  1 LOCAL_EXOG IS FILE,
    2 EX_REC (1:30) IS RECORD
     3 (II,GOV) ARE FIELDS (DEC FLOAT);
8  1 TR_IN IS FILE,
    2 TR_IN_RECORD (1:30) IS RECORD
     3 (VM,PX) ARE FIELD(DEC FLOAT);
9  1 TR_OUT IS FILE
    2 TR_OUT_RECORD(1:30) IS RECORD
     3 (VX,PM) ARE FIELD(DEC FLOAT);
10  1 LOCAL_SOL IS FILE,
     2 SOL_RECORD (1:30) IS RECORD
      3 (CONS,INV,GDP) ARE FIELDS (PIC'BB(6)9.V(6)9');

   /*DATA PARAMETERS*/
11 SIZE.SOL_RECORD = PD_SIM;
12 SIZE.TS_RECORD = PD_TS;
13 T IS SUBSCRIPT;

   /* EQUATIONS */
14 CONS(T) = IF T > 1 THEN C(1) + C(2)*GDP(T)+ C(3)*CONS(T-1)
                      ELSE TS_CONS(T+DEL);
15 INV(T) = IF T > 1 THEN C(4) + C(5)*GDP(T)+ C(6)*GDP(T-1) +
                          C(7)*II(T)
                          ELSE TS_INV(T+DEL);
16 GDP(T) = IF T > 1 THEN CONS(T) + INV(T) + VX(T)+ GOV(T) - VM(T)
                          ELSE TS_GDP(T+DEL);
17 VX(T) = IF T > 1 THEN C(8) + C(9)*PX(T)+ C(10)*GDP(T-1)
                          ELSE TS_VX(T+DEL);
18 PM(T) = IF T > 1 THEN C(11) + C(12)*PM(T-1)+ C(13)*VM(T)
                          ELSE TS_PM(T+DEL);

   /*EQUATIONS DEFINED EXTERNALLY*/
19 TR_IN_RECORD(T) = DEPENDS_ON(TR_OUT_RECORD(T));
```

FIGURE 9. Specification of a Reduced Econometric Model of a Country.

The header at the top of the figure identifies the name of the program module to be generated (A), and the names of the five source and two target files. The description of the organization of the files follows. The variable sizes of dimensions of data arrays are defined in data parameter equations (statements 11 to 13). Namely, the size or range of the lowest order (right most) dimension of the

structure TS_RECORD is equal to PD_TS, and the number of repetitions of the local solution records, i.e. the periods of simulation, is equal to PD_SIM. T is a subscript which denotes the period number of the simulation. The specification concludes with the five econometric equations in statements 14 to 18. The values of the variables for the periods prior to start of simulation (T<=DEL) come from the Time_series file. Otherwise they are defined by the respective expression.

An examination of the equations in Figure 9 reveals that statements 15 and 16 form a set of simultaneous equations. If the external dependency equation 19 is included, it forms another set of simultaneous equations with statements 17 and 18. The equations which specify in detail the external dependency are in external modules in the global configuration. The cooperation of the Configurator and the MODEL Compiler is necessary in implementing a solution process. This is further discussed in the next chapter.

6.2 CONFIGURING A MULTI-ECONOMETRIC MODEL SYSTEM

There are two problems in synthesizing selected individual country models. First, the economies of the selected countries are also highly interdependent on the economies of other countries not included in the study, and these interdependencies must be added. Second, the data in the communicating files must be compatible in meaning as well as structure. For instance, the periodicity of the time series variables (i.e. quarterly, annual, etc.) and currency units may vary from model to model. Also the export variable (VX) in each model must be disaggregated to determine the appropriate portions of the entire export of one country which become imports to each of the other countries. These portions are computed in Project LINK on the basis of the most recent bilateral trade share statistics. The portions of exports from each country to one country are summed to form its total import (VM).

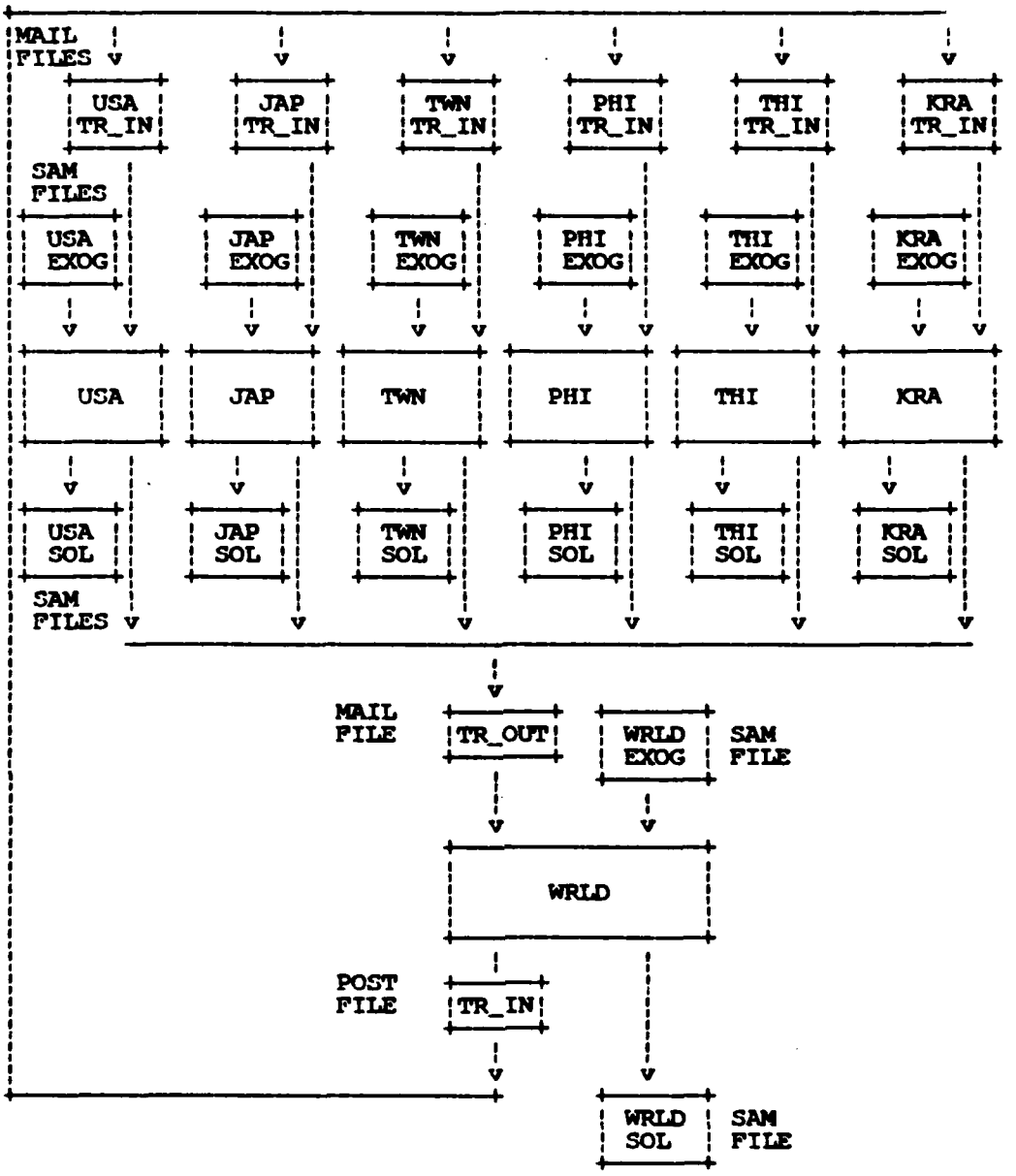


FIGURE 10(a). Configuration Network For Pacific Basin Model

The economists engaged in the study composed an interface model, which performs the above two translations. It incorporates a reduced model of the economies of all the countries not directly included in this study, to produce the data on imports to the countries in the study. It also incorporates the trade share calculation to define aggregate imports for each country as the sum of appropriate portions

of exports from all the other countries. This interfacing function forms an additional module called WORLD. It can be viewed as a case of allocation of consumable resources - with source data on exports viewed as requests for target data on imports. The WORLD model is extensive and is omitted here in the interest of brevity.

The configuration synthesizing the countries in the region is shown in Figure 10(a).

```
CONFIGURATION: PACIFIC;

F: USA_EXOG, USA_TR_IN ORG: MAIL
   -> M: USA
   -> F: USA_SOL, TR_OUT ORG: MAIL;

F: JAP_EXOG, JAP_TR_IN ORG: MAIL
   -> M: JAP
   -> F: JAP_SOL, TR_OUT;

F: TWN_EXOG, TWN_TR_IN ORG: MAIL
   -> M: TWN
   -> F: TWN_SOL, TR_OUT;

F: PHI_EXOG, PHI_TR_IN ORG: MAIL
   -> M: PHI
   -> F: PHI_SOL, TR_ OUT;

F: THI_EXOG, THI_TR_IN ORG: MAIL
   -> M: THI
   -> F: THI_SOL, TR_OUT;

F: KRA_EXOG, KRA_TR_IN ORG: MAIL
   -> M: KRA
   -> F: KRA_SOL, TR_OUT;

F: WRLD_EXOG, TR_OUT ORG: MAIL
   -> M: WRLD
   -> F: WRLD_SOL, TR_IN ORG: POST;

F: TR_IN
   -> F: USA_TR_IN,JAP_TR_IN,TWN_TR_IN,PHI_TR_IN,KRA_TR_IN,THI_TR_IN;
```

FIGURE 10(b). Configuration Specification of Pacific Basin Model

This cooperative computation scheme assumes the use of a computer network. Each model is computed in the computer of it's respective "owner" organization. Solutions are conducted in parallel in the respective computers. The distributed approach needs not be justified by reductions in cost of realizing the computation, but by the convenience to the developers which contributes to an improved system

through ready availability of global and local expertise and by shortening of development time. The main gains are in rapid development of an extensive experimentation, in depth evaluation and better reformulation which lead to a more realistic simulation of economy.

Although the centralized labor for synthesis has been eliminated, there was still a need for individuals who have global understanding of the entire system. As shown they are needed for formulating bi-lateral or multi-lateral contracts, and for selecting modules and files in composing a global system. There may be a number of simultaneous global views, represented by respective configurations formulated at the same or different locations.

CHAPTER 7

DISTRIBUTED SOLUTION OF SIMULTANEOUS EQUATIONS

As already mentioned in Section 4.3, the MODEL compiler recognizes maximally strongly connected components (MSCC) in an array graph and performs analysis and design of solution methods. Essentially it incorporates in the produced program a Gauss-Seidel iterative solution method [Greenberg, 81]. The success of the computation depends on convergence of the solution. The user may optionally specify convergence conditions and maximum number of iterations, otherwise the default value is used (presently 100 iterations). The generated code includes printing of a defined error message if convergence in not attained.

In the above example there is one cycle which is local to each module, consisting of equations 15 and 16 and the variables INV and GDP (see Figure 9). There is another MSCC, much more complicated, that involves all the modules and in each module it includes equations 17 and 18, the external dependency equation 19, the variables PX,VM,VX,PM and the records TR_IN_RECORD and TR_OUT_RECORD. The existence of an external dependency in a MSCC in a given module means that there are other modules which participa e in the solution. These modules are not known at the time that a program is generated by the MODEL Compiler. Therefore it was necessary to devise an algorithm to be incorporated in each of the modules to cooperate in the solution with external modules, without knowing their identity, number or interconnection pattern.

The method applied by the MODEL Compiler is as follows. In each module the MSCC with external dependencies includes record nodes. They are represented in the generated program by read or write operations. All the other variables in the MSCC in an individual module are solved iteratively locally. The solutions of each module of the external variables (VX,PM) are communicated to respective other modules repeatedly until global convergence is attained. This is illustrated in Figure 10(a) showing the circular connection of module. Results of each global iteration are being communicated between modules by the respective reading and writing operations. The WORLD module uses all the values of export (TR_OUT) from (I-1)th iterations of the iterative solution for evaluation of values of import (TR_IN) for Ith iteration.

In the case of an iterative solution involving a number of modules, the MODEL Compiler also generates a protocol in the produced program to determine when overall convergence, or excess of the number of iterations, has been attained, and the iterative solution should be terminated. Such protocol is generated locally in each module program without knowledge of other modules involved. The problem is similar to that of distributed termination [Dijkstra, 83; Francez and Rodeh, 82; Topor, 84]. In our solution however we want to add termination data only to records being exchanged between modules. Consequently, we may not assume that new communication channels can be added as in [Dijkstra, 83] or that existing channels are bilateral as in [Francez and Rodeh, 82; Topor, 84]. The graph, although never constructed or known explicitly, consists of nodes representing modules and directed edges which represent the records being exchanged between modules.

The algorithm is incorporated in the module programs. Each record containing values of variable from one module to another has a token added to it. The network may be viewed as a directed graph, where the modules form nodes and where the tokens are propagated along the edges. The network forms a maximally strongly connected component as illustrated in Figure 10(a). Each node determines the value of the

token sent either as equal to the minimum on the values of the incoming tokens plus 1 or if it has not locally reached convergence (or exceeded the maximum number of iterations) then 1. This information is propagated, one node on each iteration, throughout the network. If the diameter of the network is D, it takes D iterations until the information reaches the furthermost module. In this manner, D+1 iterations after all the module have reached convergence, the value of the tokens would be D+1 in all the modules and they all terminate iterations simultaneously.

Description and derivation of the termination algorithm are given in section 16.2 (Part III). Implementation techniques used to incorporate the algorithm is described in section 16.3 (Part III).

CHAPTER 8

CONCLUSION AND FUTURE RESEARCH

As mentioned earlier, this dissertation intended to make two points. First that the use of a new class of languages in concurrent programming, which we characterize as very high level, equational, definitional, nonprocedural or dataflow, is natural and effective. The effectiveness is justified by the number of definitional statements in the first example, especially the R module. The R module uses 20 statements for defining the "maximum claim" algorithm (for resource allocation problems in general). The naturalness can be justified by the second example which takes almost literally the econometric equations from the LINK project and the MODEL compiler translates them into a distributed operational system. Also, the studies done by [Cheng, 83] shows evidences of naturalness and effectiveness in other aspects.

Second, that an automated program design methodology can verify the specification and translate the specification into efficient concurrent computation using existing computer technology. The presentation through use of characteristic examples was intended to convey the flexibility and practicality of the programming approach in present day applications. The current implementation relies on modern operating systems and computer architecture, currently VMS and VAX-750. Theoretically the two system can be implemented on UNIX, or any other machines.

In reviewing the new concepts in computer architecture we believe that a similar methodology would be devised in the future for

implementing parallel processing in new generations of computers. The independence of the language from the object computer architecture makes it a good candidate for use for future computer architecture. The dataflow approach exposes the possible parallelisms and can be used for programming, for instance, for dataflow computer machines [Gokhale, 83].

As noted, the research described in this dissertation has still unsolved problems. The future research under this main direction is summarized below:

i) the checking of convergence and parallel processing of recursive functions,

ii) development of a system that will evaluate timing between receiving and sending of records, to verify if real time system timing constraints are satisfied, and to modify the design if not [Tseng, 83],

iii) further extend the semantic checking of a specification on MODEL specification level and configuration level,

iv) further automate program generation by generating external dependencies automatically,

v) more effective file organizations for general applications,

vi) more flexible definition of module execution time and more verification on execution sequence of a configuration, and

vii) automatic generation of query sub-system by the Configurator to facilitate on-line configuration information retrieval by using some functional languages, such as PROLOG or LISP.

Also, the entire problem of automatic partitioning of a computation into modules and files to attain a high degree of parallelism is an as yet unsolved one which is assuming major importance in Computer Science.

PART II

THE CONFIGURATION SYSTEM

CHAPTER 9

INTRODUCTION TO THE CONFIGURATOR

The function of the Configurator is to synthesize program modules and data files that may be operated concurrently and/or distributed geographically. Establishing communications between modules and integrating them into a structured system remains a complex and error prone task. We refer to this task as to the configuration of a system. The Configurator assists the user to verify the consistency and completeness of the system and to synthesize the components into an integrated system.

9.1 THE LANGUAGE - CSL

CSL, standing for Configuration Specification Language, is designed for describing a network of modules connected through data files. It is a "path language", because it describes a network by listing its paths.

CSL aims at a broad range of applications, which include configurations of sequential, concurrent/sequential, and interactive systems. To avoid repeating paths in a CSL specification, only nonredundant paths have to be specified. The system automatically determines all the paths whose existence can be derived from combining those that are explicitly specified by the user.

9.2 THE PROCESSOR - CONFIGURATOR

The Configurator is the compiler of CSL. As mentioned in Chapter 3 (Part I), the Configurator has five functions: checking the input CSL specification, scheduling execution of modules, evaluating diamters of strongly connected components, generating JCL and PL/I programs and generating user system documentation.

Chapter 11 of this Part is devoted to presenting the working principles and translation mechanisms of the Configurator.

CHAPTER 10

THE CONFIGURATION SPECIFICATION LANGUAGE

In this chapter, we give the syntax and semantics of CSL statements for describing a system configuration.

10.1 OVERALL GRAPH DESCRIPTION

A multi-module system may have sequential, concurrent and/or distributed components. It is represented as a network of modules and files. It can be visualized as a graph where the modules and files are nodes and the relations of modules consuming or producing files are edges. A CSL description of a network consists of a set of statements describing the paths in the network. Each path is defined in a statement as a chain of nodes connected by edges.

The overall structure of a configuration specification is:

```
<CONFSPEC>::= CONFIGURATION: <IDENTIFIER>
              [( DIAM : <INTEGER> )] ; <STATEMENTS>
<STATEMENTS>::=<STATEMENT> [;<STATEMENT>]* ;
  <STATEMENT> ::=<PATH_STATEMENT> | <SYNONYM_STMT>
```

FIGURE 11. Syntax structure of a CSL specification

A specification has a number of statements, first it has a name - an identifier, which is a string of letters and digits, beginning with a letter. The length of an identifier is limited to eight characters.

The name of the configuration may be optionally followed by a parenthesized parameter of the configuration, called diameter. The diameter is needed only in the cases where there exist cycles in the

configuration that represent communicating modules that jointly perform a solution of simultaneous equations. Namely, where the simultaneous equations are distributed over all the modules in respective cycles. In such a case, it is necessary to evaluate the maximum distance between modules in order to have orderly termination of the iterative solution. The Configurator calculates the diameter automatically for each strongly connected component (i.e. cycles) in the configuration graph. The user may have better insight and wish to over-ride the automatically evaluated diameter. This is further discussed in section 10.8.

Next, there are two types of statements - to describe paths and synonyms.

<PATH_STATEMENT> ::=<NODES> [-> <NODES>]*

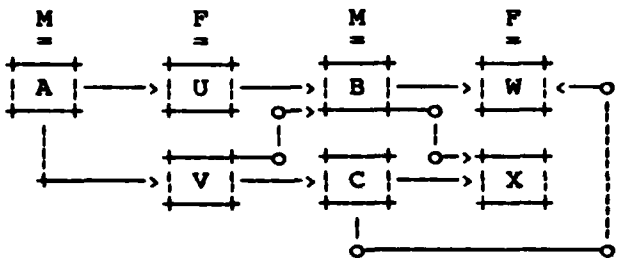
FIGURE 12. Syntax of a path statement

The path statement is used to describe a path in the configuration. The path statement consists of a chain of lists of nodes connected by the symbol "->" which represents an edge. A list consists of one or more nodes. A path statement may contain only one list of nodes. In such a case, the statement merely declares the existence of nodes.

Example 10.1.

```
M: A -> F: U,V -> M: B -> F: W,X;
F: V -> M: C -> F: W,X;
```

The above CSL statements describe the inter-connection between three module nodes(A,B,C) and four file nodes(U,V,W,X). The corresponding graph is shown below.



We will say that the files coming into a module as being "consumed" by the module and the files going out of a module as being "produced" by the module.

A more detailed description of the path statement is given in the following section.

The SYNONYM statement is used to unify differently named nodes into one node, when these nodes actually correspond to only one physical entity in a configuration.

The syntax diagram of a synonym statement is as follows.

```
<SYNONYM_STMT> ::= S: <NAME>, <NAME> [, <NAME>]*;
     <NAME>     ::= [ <IDENTIFIER>.] <IDENTIFIER>
```

FIGURE 13. Syntax of a Synonym Statement

Example 10.2.

S: A, B, C;

specifies that nodes A, B and C correspond to one physical entity. A more detailed description of the SYNONYM statement is given in section 10.7.

10.2 NODES IN A PATH STATEMENT

The syntax diagram of a node description is:

```
<NODES>         ::= <MODULE_NODES> | <FILE_NODES>
  <MODULE_NODES> ::= M : [+] <IDENTIFIER> [<M_ATTRIBUTES>]
                         [,[+] <IDENTIFIER> [<M_ATTRIBUTES>]]*
  <FILE_NODES>   ::= F : [<QUALIFIER>.]<IDENTIFIER> [<F_ATTRIBUTES>]
                       [,[<QUALIFIER>.]<IDENTIFIER> [<F_ATTRIBUTES>]]*
```

FIGURE 14. Syntax of a configuration node

There are two kinds of nodes in a configuration: MODULE nodes and FILE nodes. A list of MODULE or FILE nodes is prefixed by M or F respectively. Furthermore, a MODULE node may be prefixed by a "+" symbol to indicate that the module's execution may be "added", i.e. initiated manually, rather than automatically.

A FILE node can also be prefixed by the producer or consumer MODULE name, to differentiate between files with the same name.

A node may have a number of optional attributes, such as type (described further), physical file name, record size and device description. When an attribute is not specified, a default value is assigned to it.

10.3 MODULE NODE ATTRIBUTES

The syntax diagram of a module node attribute is:

```
<M_ATTRIBUTE> ::= [<M_TYPE>] [<PHYSICAL_NAME>]
    <M_TYPE>        ::= ( TYP : <MTP> )
    <MTP>           ::= MDL | GRP | MAN
    <PHYSICAL_NAME> ::=  = [<NETWK_ADDR>][<DIRECTORY>][<P_NAME>]
                           [<SUFFIX>] [<VERSION>]
        <NETWK_ADDR>  ::= <IDENTIFIER> ::
        <DIRECTORY>   ::= ( <IDENTIFIER> [.<IDENTIFIER>]* )
        <P_NAME>      ::= <IDENTIFIER>
        <SUFFIX>      ::= . <IDENTIFIER>
        <VERSION>     ::= <INTEGER>
```

FIGURE 15. Syntax of an attribute of a module

Underlined is the default attribute.

10.3.1 MODULE TYPE ATTRIBUTE (M_TYPE)

A MODULE node, in general, corresponds to a user defined function which can be a program, a terminal, or a set of programs and terminals. A module may consume and produce file(s). A module of type MDL, corresponds to an executable program produced either by the MODEL compiler, or by a programmer. A node of type GRP is a sub-system which consists of a group of lower level modules which may also consume and produce file(s). It is represented as a single module node. This feature can be used in developing large systems to represent a hierarchy existing inside a system. The use of GRP type nodes can also improve the readability of a CSL specification.

The analysis and scheduling phases in the Configurator assume that all the modules re ATOMIC, meaning that they acquire all their files at the beginning and release them at the end of processing. This is generally true for the programs generated by the MODEL compiler. However, a GRP node is generally not ATOMIC, in that its constituent module nodes acquire and release files according to the way they are scheduled by the Configurator. The use of non-atomic module nodes may cause loss of efficiency in scheduling and loss of the ability for conducting various consistency checking. The alternative for the user is to provide more detailed information by replacing a non-atomic node by its more elementary atomic constituents.

A node representing manual interactions - MAN, corresponds to a terminal. A MAN module consumes a file produced by another module, displays it on the screen of the terminal and produces a file corresponding to data keyed in by the terminal operator. Introduction of the MAN type allows represent also manual interactions in a configuration. The MAN type informs the Configurator to generate special commands for connecting the I/O channels to the keyboard and screen of a terminal.

10.3.2 PHYSICAL NAME ATTRIBUTE (PHYSICAL_NAME)

The ‹PHYSICAL_NAME› attribute binds the logical name given by the identifier of a node to a physical name of a program or a data file existing in the network. The syntax of the command language for Digital Equipment Corporation's VAX/VMS operating system is followed here.

The physical name provides an address in a computer network, and in a user's directory. It may optionally have a suffix indicating the contents of the physical entity (explained below) and a version number.

‹NETWK_ADDR› is the address in a computer network. The addresses should be known to the VMS network communication package which can then setup communications between the specified sites. The default is the local site where the Configurator is run.

‹DIRECTORY› is the name of a directory or sub-directory of user's account. The default is the root directory.

‹P_NAME› specifies a physical entity in the directory. It may differ from a logical name adding flexibility to naming in CSL. ‹P_NAME› is an identifier up to 10 characters long. The default ‹P_NAME› is the logical name.

‹SUFFIX› is a three character name attached to the P_NAME. It may be used to indicate the type of data stored in the identified file. The default is " "(blank). A user can add an arbitrary suffix to a P_NAME. Some of the predefined suffix names and their meanings in VMS are summarized below.

```
PLI —› PL/I
FTN —› FORTRAN
PAS —› PASCAL
DAT —› Data file
LIS —› Listing file produced by compilers
COM —› VMS command program
```

<VERSION> is an integer and is used to indicate a particular version of a file on a computer. On VAX, the VMS retains old versions of a file which have to be explicitly deleted. One can address different versions of a file by specifying version numbers. The default is the most recent version of the file.

Example 10.3.

```
M: P1=UPENN-750::(YUAN.TEMP)TEST.PLI;10
```

specifies that a module node with logical name P1 corresponds to a file in a computer network with a computer address "UPENN-750", in directory [YUAN.TEMP] bounded to the physical name TEST with suffix PLI and version number 10.

10.4 FILE NODE ATTRIBUTES

The syntax diagram of the attributes of a FILE node is as follows.

```
<F_ATTRIBUTE>::= [<F_TYPE>] [<PHYSICAL_NAME>]
                 [<RECORD_SIZE>] [<F_DEVICE>]
    <F_TYPE>      ::= ( TYP : <FTP> )
    <FTP>          ::= SAM | ISAM | POST | MAIL
    <RECORD_SIZE>::= RS: <INTEGER>.
    <F_DEVICE>    ::= DEV: DISK | TAPE
```

FIGURE 16. Syntax of attribute description of a file node

A FILE node corresponds to an entire logical data structure consumed or produced by a module.

10.4.1 FILE TYPE ATTRIBUTE (F_TYPE)

The F_TYPE attribute determines the organization of a file and thus the use of the file in the network.

A SAM file is a data file existing as one entity. Namely, if the file is exchanged between modules, the file must be completely

produced by a module before it can be consumed by other modules. In a path

M:M1->F:F1 (TYP:SAM)->M:M2,

let the starting time of the modules M1 and M2 be denoted by ts1 and ts2, and ending time by te1 and te2, then the path implies te1 $\leq$ ts2.

In the current version of the system, it is implemented as a sequential file residing on disk or tape, and thus a SAM file is retained even after it has been consumed. Whether a file is on disk or tape must be defined in the F_DEVICE attribute. If the file is on tape, a number of modules may consume it only in sequence (with a rewind in between). If the file is on disk, it may be consumed concurrently by a number of modules. SAM is the default value of the F_TYPE attribute.

An ISAM file is a set of data whose individual units (records) can be consumed and produced concurrently by many modules. Each record, except when it is redefined (updated) by producing modules, is retained in the file even after it has been consumed. There are no timing restrictions on modules connected through an ISAM file. An ISAM file is implemented as an indexed sequential file indexed by keys, randomly accessible and can reside only on disk. An ISAM file may have a number of separate versions ("older" and "newer") represented by same named (or synonymous) ISAM file nodes connected by edges, indicating sequentiality between producing and consuming respective versions. In a path:

M:M1 -> F:F1 (TYP:ISAM) -> F:F2 (TYP:ISAM) ->M:M2,

then F1 and F2 are two versions of the same file and te1 $\leq$ ts2.

This is described further in section 10.5 which discusses the semantics of the edges.

A MAIL file represents data being communicated between one or several concurrent producing and/or consuming modules. Units of communication called records are produced one or several at a time, and queued in the order of their times of production. The records are

consumed by the consuming module in the same order. Thus a MAIL file serves also as a queue when there are several producer and/or consumer modules. The production and consumption of records may be concurrent and is automatically synchronized, therefore synchronization need not concern the user of CSL. In a path:

M:M1 -> F:F1 (TYP:MAIL) -> M:M2,

it is required that ts1 > ts2 & te2 < te1.

Requirements of compatibility between specifications of interfacing files are described in section 10.6.

A POST file also represents data being communicated in record units, similar to a MAIL file. However, a POST file must consist of records containing addresses of their destinations. Each record is automatically delivered to the indicated destination - a MAIL file. A POST file is produced by only one module but it can be connected to any number of destination MAIL files being consumed by different modules. This is further explained in connection with discussion of the edges. The POST file records also have a compatibility requirement described in section 10.6.

The address provided in records in a POST file must be the physical name of the destination file. The construction of a physical name is as follows.

i) For MAIL files that are source of a module which is not prefixed by "+", the physical name is: ‹logical name›S_MBX

ii) For MAIL files that are sources of a "+" module, the physical name is : ‹logical name›S_MBX‹creation time›.

In this way, the system can distinguish the different instances of a "+" module which uses the same module and file names. In the MODEL specification, the user can use the function PHYS_NAM(logical name), which returns a physical name bound to the logical name at runtime, to define an address field in a POST file.

The selection of file types is based on the requirements in the configuration for concurrency, distributed operations or supply of data. These are discussed in further detail in section 10.6.

Note on implementation:

The file types discussed above can be implemented in many ways, depending on the operating system features available on the target computer. The SAM and ISAM correspond naturally to sequential and index sequential file organizations supported by commercially available operating systems. Under VAX-VMS, the MAIL and POST files have been implemented via the mailbox facility. Under different operating systems the implementation might be totally different.

10.4.2 PHYSICAL NAME ATTRIBUTE (PHYSICAL_NAME)

This attribute has the same syntax and semantics as the physical name attribute of a MODULE node, described in section 10.3.

10.4.3 RECORD SIZE ATTRIBUTE (RECORD_SIZE)

This attribute is only required for MAIL files. It defines the maximum size of records. The default is 300 characters (bytes).

10.4.4 FILE DEVICE ATTRIBUTE (F_DEVICE)

This attribute is used only for SAM files. The default is DISK.

This attribute allows the Configurator to determine whether a file can be consumed concurrently (DISK files) or have to be consumed sequentially (TAPE files).

10.5 EDGES

An edge represents consumption, production or causality relationships between two nodes. The direction of an edge is indicated by the "->" symbol in a CSL statement.



This section discusses the meaning of an edge, depending on the attributes of its source and target nodes.

i) If the source is a module node and the target is a file node, an edge indicates that the file is produced by that module.

ii) If the source is a file node and the target is a module node, an edge means that the file node is consumed by that module.

iii) If the source is a POST file node and target is a MAIL file node, the edge represents the distribution of records. The edge means that the MAIL file may receive records from the POST file. The MAIL file must be the source file of some module node(s). This kind of an edge is required to be drawn from a POST file to each potential MAIL file destination.

iv) If the source and target are both ISAM files, the edge indicates that the source is an "older" version of an ISAM file and the target is a "newer" version of that file. The "newer" version is available only after the production and/or consumption of the "older" version has been completed. This kind of edge allows a user to indicate the sequential access to an ISAM file. Two ISAM files or a long chain of progressively newer versions of an ISAM file may be connected in this way.

Depending on the file type attribute, there are also constraints on the number of edges that can be associated with each node. This is depicted in the following table.

FILE PRODUCTION AND CONSUMPTION RULES

TYPE	# OF PRODUCERS	# OF CONSUMERS	REMARKS
SAM	n=0,1	m>=0	n + m > 0
ISAM	n>=0	m>=0	n + m > 0
MAIL	n>=1	m>=0	n + m > 0
POST	1	m>=1	———

MODULE PRODUCTION AND CONSUMPTION RULES

TYPE	# PRODUCED	# CONSUMED	REMARKS
MDL,GRP	n>=0	m>=0	n + m > 0
MAN	n=0,1	m=0,1	n + m > 0

TABLE 1. File and Module Production and Consumption Rules

10.6 REQUIREMENTS OF CONNECTING FILES

The user of CSL must select the type of a file which connects modules based on the functional requirements of the configuration as a whole. The selection of the file type must be reflected both in the individual producing or consuming module specifications and in the configuration specification. The functional requirements concern concurrency, distributivity and sequentiality in operation of modules (section 10.6.1). In addition, there are requirements of compatibility in file structures as specified in the specifications of the producing and consuming modules (section 10.5.2).

Finally, there are restrictions on modules which are initiated manually (section 10.6.3).

10.6.1 CONCURRENCY, DISTRIBUTIVITY AND SEQUENTIALITY

The requirement and choice of file types should be guided by the following:

i) If a copy of a file must be retained then only SAM or ISAM types files may be used. A SAM file is used when the producer module must precede entirely the consumer. And an ISAM file may be used when such a constraint does not exist.

ii) If the connected modules are to be initiated sequentially, one preceeding the other entirely, the relationship among them may be expressed as follows:

   a) The path between the predecessor and successor modules includes a SAM file, or

   b) The path between the predecessor and successor modules includes a pair of same named (or synonymous) ISAM files connected by an edge in the direction from the predecessor to the successor.

iii) If the connected modules may be activated concurrently and/or distributively, then they must be connected by either MAIL or POST connected to MAIL.

Modules are generally scheduled to be executed as early as possible, concurrently or otherwise, subject only to other sequential dependencies in the configuration graph. Thus, the user of CSL must consider whether certain modules may be initiated at the same time or one preceding another. MAN type modules must be concurrent with the modules to which they are connected via files. Requirements of sequentiality are usually imposed by the outside environment (schedule of work of people, etc.). Otherwise, it is generally more efficient to operate modules concurrently and, in cases that do not involve other considerations, it is preferable to use MAIL files.

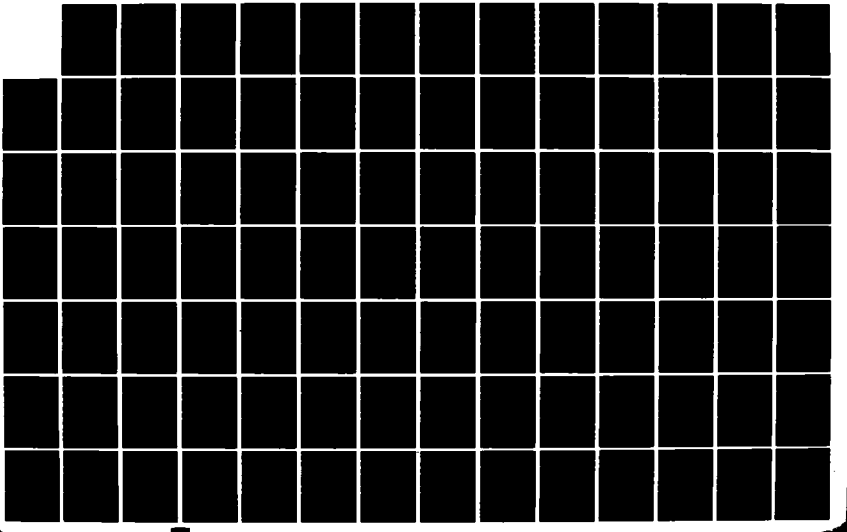
Distributed modules can only exchange messages through MAIL and POST files. Consequently, every SAM or ISAM file consumed or produced by a module must be located in the same node(computer) in a network with the module. Modules exchanging information through SAM or ISAM, if resided on different nodes in a network, will signal an error.

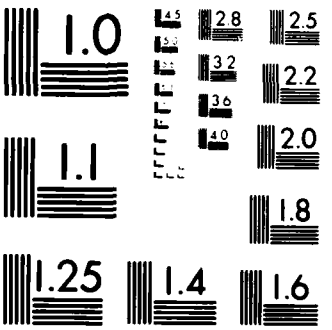
10.6.2 COMPATIBILITY OF CONNECTING FILES

The definition of data structures of a connecting file must be included in the MODEL specifications of all its producer and consumer modules. These data structure declarations in the different modules

AD-A150 009 VERY-HIGH LEVEL CONCURRENT PROGRAMMING(U) MOORE SCHOOL OF ELECTRICAL ENGINEERING PHILADELPHIA PA DEPT OF COMPUTER AND INFORMATION SCIENCES Y SHI DEC 84 2/3

UNCLASSIFIED N00014-83-K-0560 F/G 9/2 NL





MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS 1963-A

may vary i some respects but otherwise must strictly agree.

i) RECOR STRUCTURES: Corresponding record structures must have the same .ength and their respective constituent groups and fields must ve the same dimensionality, range, data type, length and scale Namely the trees representing respective record struc res in different module specifications must be the same. Howev , the respective records, groups and fields may be named diffe ntly.

ii) FILE RUCTURES: The order of different record structures in a conne ing file (from left to right in the file tree) must be the same all the specification of the modules connected by the file. Also the total numbers of records with respective data struc res must be the same. Note that the number of records need ot be constant. The number of records may also be denoted for e h dimension by END-OF-FILE marker or by variables (with END SIZE prefix). It is not required that the number of dimen ons of respective records be the same, but their total numbe must be the same. The END-OF-FILE marker must not be used to de te the size of a vector of records in a MAIL file, other means such as a constant or END or SIZE prefixed variables, may be us l.

iii) VIRTU DIMENSIONS OF RECORD STRUCTURES: To achieve concurrent opera on of modules connected by MAIL files (or POST connected to MA), the producer module has to produce one record, or a group f records, at a time and these records have to be consumed also e, or a group of them at a time. This requirement corre onds in the MODEL system to having virtual dimension at the h hest order, and possibly immediate lower order, record dimen ons. Information about the types of dimensions for a speci cation is included in the MODEL reports. It is the respc ibility of the CSL user to verify the above said requi ments to be satisfied.

10.6.3 MANUAL INITIATION OF MODULES

The modules that are to be initiated manually must be prefixed in a CSL specification with the "+" sign. It is common in large systems that the number and duration of operation of some modules is not known in advance. This is typical in modules that are connected to a MAN type of module, i.e. where a human operator of a terminal communicates with a module. Such a module is then initiated by the operator. The number of such modules and when they are initiated depends on the number and schedule of work of the terminal operators. There are several requirements of a module prefixed by "+" as follows.

i) SAM files: The module may produce or consume SAM files provided that they are not connected to other modules.

ii) concurrent operation: The module must operate concurrently with other modules as follows:

   a) it may produce files only of type MAIL (or POST connected to MAIL) or ISAM which may be connected to other modules

   b) it may consume files only of type POST connected to MAIL or ISAM which may also be connected with other modules.

10.7 SYNONYM STATEMENT

A synonym statement is used to identify the equivalence between logical names. Synonymous names correspond to a single physical entity.

Synonymous names must be all of MODULE or all of FILE, and must have the same or complementary attributes. Synonymous file names may be prefixed by the producing or consuming module name.

The synonym relation is symmetric, transitive and reflexive, i.e. if S: A,B; and S: B,C; are two synonym statements, then S: B,A;, S: B,C; and S: A,C; are implied. Also for every node X in a configuration, S: X,X; is always assumed.

10.8 OPERATING THE CONFIGURATOR

10.8.1 INVOKING THE CONFIGURATOR

To activate the Configurator, a user calls: "@RCONF <CSL specification name>". This executes the following VMS command procedure:

```
$ASS ERMSGE.DAT ERMSGE      /* Connect the ISAM error msg file */
$ASS 'P1'.CNF SAPIN         /* Connect P1(CSL specification)   */
$ASS SYSOUT.DAT SYS$OUTPUT  /* Connect the timing/trace file   */
$ASS SYSIN.DAT  SYS$INPUT   /* Connect the parameter file      */
$RUN CONF                   /* Activate the Configurator       */
$DEASS ERMSGE               /* Disconnect error msg file       */
$DEASS SAPIN                /* Disconnect CSL spec file        */
$DEASS SYS$OUTPUT           /* Disconnect screen output        */
$DEASS SYS$INPUT            /* Disconnect keyboard input       */
```

10.8.2 I/O FILE NAMING CONVENTIONS

NAME CONVENTION	I/O	COMMENT
"name1".CNF	I	A file containing a CSL specification named "name1"
SYSIN.DAT	I	Parameter file
ERMSGE.DAT	I	Syntax error ISAM file
RPT.DAT	O	Report file
SYSOUT.DAT	O	Configurator timing and debugging message file
CONFERR.DAT	O	Error message file
"name1".COM	O	Main JCL program file
"name1".PLI	O	Mailbox creation program
"name1"D.PLI	O	Mailbox deletion program
"namei".COM	O	Individual JCL program file

TABLE 2. I/O File Name Conventions of the Configurator

The Configurator generates N+1 JCL programs, where N equals to the number of modules in a configuration. Each individual JCL program is named after the corresponding module name ("namei" in the above table) in the configuration and the main JCL program is named after the name of the configuration.

The report file (RPT.DAT) may contain up to seven reports generated by the Configurator according to selected options (see section 11.3.5). The system timing and debugging message file

(SYSOUT.DAT) contains the timing of each processing stage, and if debug option has been selected, the debugging messages from the Configurator. The debugging messages are provided for debugging the Configurator.

10.8.3 PARAMETERS TO THE CONFIGURATOR

The Configurator uses a parameter file (SYSIN.DAT) which is used to select options to perform or not perform certain functions of the Configurator.

The parameters are summarized below.

PARAMETER	MEANING
1. NT/TRACE	—turn off/on the debugging messages
2. NC/GENCODE	—turn off/on the code generation switch
3. LST/NLST	—do/do not list the CSL specification
4. GF /NGF	—do/do not generate the configuration graph(Gf) report
5. XREF/NXREF	—do/do not generate the CSL cross reference report
6. MSCC/NMSCC	—do/do not generate MSCCs in Gf
7. GE/NGE	—do/do not generate the extended component graph(Ge) report
8. MFREF/NMFREF	—do/do not generate the module-file cross reference report
9. JCLS/NJCLS	—do/do not list the JCL and PL/I programs.

TABLE 3. Parameters to the Configurator

The defaults are underlined. The parameters can be positioned in SYSIN.DAT in an arbitrary order in a line and separated by a blank or any other delimiters. For example, the following can be the content in a SYSIN.DAT: "XREF,MSCC,GE,GENCODE".

CHAPTER 11

THE CONFIGURATOR

The Configurator is in fact the CSL compiler. It accepts, as input, a specification in CSL and produces executable PL/I programs for setting up communications between modules, command programs for executing the configuration and documentation of the configuration.

11.1 STRUCTURE

The main procedure of the Configurator is described in section 11.3.

The Configurator processes a CSL specification through the following stages:

[1] Syntax analysis and construction of the configuration graph (section 11.4)
[2] Completeness analysis (section 11.5)
[3] Sequence checking (section 11.6)
[4] Scheduling (section 11.7)
[5] MSCC diameter evaluation (section 11.8)
[6] Code generation (section 11.9)
[7] Configuration documentation (section 11.10)

The control flow of the Configuator is depicted in Figure 17.

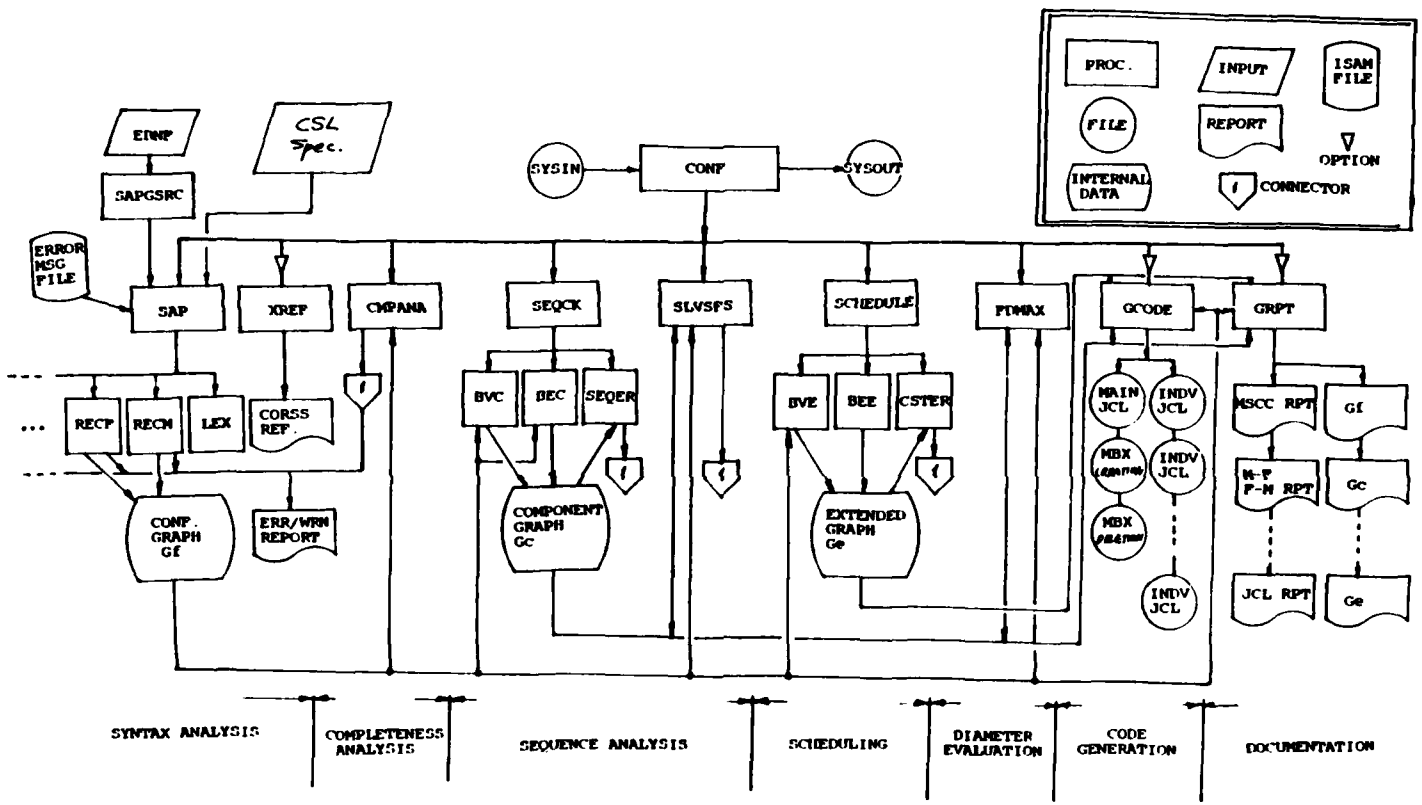


FIGURE 17. The Processing Stages of The Configurator

11.2 PRODUCTS

The Configurator produces a number of outputs. They consist of a set of configuration programs and documentation. The set of programs is in two languages: JCL and PL/I. The reports and programs are listed in the Documentation and Code Generation columns at right of Figure 17. The JCL programs can be categorized into two groups:

i) A main JCL program - which creates mailboxes and submits individual JCL programs into an available operating system (currently VMS on VAX-750) and synchronizes the termination of the JCL program with the terminations of all the modules in the configuration.

ii) Individual JCL programs - one for each module node in the configuration graph. Each individual JCL program synchronizes its sequential predecessor(s), connects the logical file names to appropriate physical files and activates a module. When the module terminates, the JCL program disconnects the files.

The PL/I programs:

i) Mailbox creation program

ii) Mailbox deletion program

The mailbox creation program is activated by the main JCL program to create necessary mailboxes before the system execution. Each individual module also creates mailbox(es) for its own needs and deletes the mailbox(es) at its completion. This allows a module to be initiated repeatedly without re-activating the mailbox creation program in the main JCL program.

The mailbox deletion program is useful in the debugging of the system. A test process(module) may fail to complete processing and the failure may occur before the mailbox deletion part in the program. The mailbox(es) associated with the module may then contain some unprocessed messages which may effect the next run. The mailbox deletion program can be used to clear up the mailboxes. Therefore it

must be activated manually in case of process failure.

Each run of the Configurator generates up to seven reports. The user can select desired reports (see Table 3).

11.3 THE MAIN CONFIGURATOR PROGRAM: CONF

The main configurator program calls the procedures for respective phases of the Configurator. The calling sequence of different processing stages are shown in Figure 17. The calling sequence of processing stages is from left to right in the figure.

11.4 SYNTAX ANALYSIS AND CONFIGURATION GRAPH CONSTRUCTION (PROCEDURE NAME: SAP)

11.4.1 THE SYNTAX ANALYZER

The syntax of CSL is described in the Extended Backus-Naur Form With Subroutine Calls(EBNF/WSC). The EBNF part defines the syntax of CSL, and subroutine calls incorporated into it facilitate semantic checking and configuration graph construction. The following is a brief presentation of the syntax analyzer of CSL.

A recursive descendent parser generator is employed in processing of the EBNF/WSC description. The parser generator reads the EBNF/WSC description of CSL and generates a Syntax Analysis Program(SAP). The SAP calls the embedded semantic subroutines while parsing CSL statements. Semantic checking, error reporting and graph construction are performed by those routines. This scheme has allowed easy modification of the syntax and/or semantics of the earlier versions of CSL.

The semantic routines can be classified into two different categories.

i) Semantic checking and graph construction

These routines recognize particular symbols, create new nodes and construct a configuration graph while SAP is parsing the CSL statements. The syntactical restrictions that are not expressed in EBNF are checked by particular routines. For example, if an identifier M is used as the name for a MODULE node and "F: M" appears in the CSL specification, the routine CK_NAME will report this as an error(see next section).

ii) Error message routines

These routines perform syntax error reporting. They are referred to in EBNF/WSC as /E(i)/, where i denotes an integer representing an error code. They are always placed before a routine that recognizes a keyword or a delimiter. SAP stacks error codes (from the corresponding reference /E(i)/) whenever it calls the routine. When SAP fails to recognize a certain symbol, the code will be on the top of the stack. SAP then pops the stack and calls a routine $FAIL which prints a predefined error message indexed by that code. The list of all the warning and error messages produced in syntax analysis can be found in Appendix C.

The same lexical analyzer (LEX) is used in the MODEL processor. LEX is called by various routines from SAP.

In the EBNF/WSC description, the semantic routine calls are enclosed in "/" signs. The following is the complete EBNF description of CSL.

```
<CONF_PROG> ::= <HEADER> [<DECLARATIONS>/CLRERRF/]*
                /STMT_FL/ <CONF_PROG>
<HEADER>    ::= CONFIGURATION: /E(10)/ <NAME> /CONFNM/
                [ DIAM : <INTEGER> ] /E(1)/ ; /LINENUM/
<DECLARATIONS> ::= <SYNONYM> /E(1)/ ; /LINENUM/
                 | <DECLARATION> /E(1)/ ; /LINENUM/ /CLRALL/
                 | @#_END#@ /ENDINP/
```

(to be continued)

```
<DECLARATION>   ::= <M> /E(4)/ : /E(2)/ <M_NAMES>
                      [ <ARROW> /E(11)/ <M_F> /STMF/]*
                    | <F> /E(4)/ : /E(2)/ <F_NAMES>
                      [ <ARROW> /E(11)/ <M_F> /STMF/]*
<M_F>           ::= <M> /E(4)/ : /E(2)/ <M_NAMES>
                    | <F> /E(4)/ : /E(2)/ <F_NAMES>
<ARROW>     ::= /ARROW/
<M>         ::= /RECM/
<F>         ::= /RECF/
<M_NAMES>   ::= <M_NAME> /ADDNAME/ [,<M_NAME> /ADDNAME/]*
<F_NAMES>   ::= <F_NAME> /ADDNAME/ [,<F_NAME> /ADDNAME/]*
<M_NAME>    ::= [ + /GETPM/] [<LOC>] [<DEVICE>] [<DIRECTORY>] /E(5)/
                <NAME> /STLNAME/
                [=/E(5)/<NAME> /STPNAME/
                [<SUFIX> [<VSN>] ]]
                [ ( /E(12)/ <KORG>: /E(6)/ <M_TYP> /STORG/ )] <CK_NAME>
<F_NAME>    ::= [<LOC>] [<DEVICE>] [<DIRECTORY>] /E(5)/
                <NAME> /STLNAME/ [. <NAME> /MDFLNM/ ]
                [=/E(5)/<NAME> /STPNAME/
                [<SUFIX> [<VSN>] ]]
                [ ( /E(12)/ <KORG>: /E(6)/ <F_ORG> /STORG/ [)]
                [ RS: /E(7)/ <INTEGER> /STSIZE/ )]] <CK_NAME>
<KORG>      ::= ORG | TYPE | TYP
<LOC>       ::= /GETLOC/
<DEVICE>    ::= /GETDEV/
<DIRECTORY>::= /GETDIR/
<SUFIX>     ::= . /GETTYP/
<VSN>       ::= . /E(8)/ <INTEGER> /STVSN/
<CK_NAME>   ::= /CKNAME/
<F_ORG>     ::= SEQL | ISAM | MAIL | POST
<M_TYP>     ::= MAN  | GRP  | MDL

<SYNONYM>   ::= S : /E(9)/ <L_NAME>/E(10)/,/E(9)/<L_NAME>
                [, /E(9)/ <L_NAME>]* /MGSYN/

<L_NAME>    ::= <NAME> /STSNAME/ [. <NAME> /MDFLNM2/ ]
<NAME>      ::= /NAMEREC/
<INTEGER>   ::= /INTREC/
```

FIGURE 18. EBNF/WSC Description of CSL

The following table contains all the semantic routines and their functions in the above EBNF/WSC description.

NAME	FUNCTION
CLRALL	Clear global and local graph pointers.
ARROW	Recognizes the symbol "=>".
RECM	Recognizes "M" as the prefix of node(s).
RECF	Recognizes "F" as the prefix of node(s).
ADDNAME	Adds the currently scanned name into the local graph.
STLNAME	Stores a logical name.
MDFLNM	Modifies the stored logical name to be prefixed.
STPNAME	Stores a physical name.
STORG	Stores organization of a file node.
STSIZE	Stores record size.
GETPM	Stores the "+" sign of a node.
GETLOC	Recognizes and stores the network address of a node.
GETDEV	Recognizes and stores a device description.
GETDIR	Recognizes and stores a directory description.
GETTYP	Gets the suffix of a physical name.

(to be continued)

STVSN	Recognizes and stores a version number.
CKNAME	Checks the currently scanned nodes's attribute.
MGSYN	Merges synonymous names into one node.
STSNAME	Stores the current synonymous name in a list.
MDFLNAME2	Modifies the node name in a synonym list to be prefixed.
NAMEREC	Recognizes an identifier.
INTREC	Recognizes an integer.

TABLE 4. Semantic Routines For CSL

11.4.2 DEFINITION OF A CONFIGURATION GRAPH

Let M and F denote two non-empty disjoint sets of elements. We interpret elements of the set M as modules, and those in the set F as files.

DEFINITION 1.

A DIRECT CONFIGURATION GRAPH is a graph Gf'=(Vf',Ef'), with a set of nodes: $Vf'=M \cup F$, and a set of edges $Vf' \in (M \times F) \cup (F \times (F \cup M))$.

For an edge $e=\langle v1,v2\rangle \in Ef'$, we will assume the following interpretation:

i) If $e \in Ef' \subset (M \times F)$ then it represents a production relationship, i.e. the file v2 is produced by module v1;

ii) If $e \in Ef' \subset (F \times M)$ then it represents a consumption relationship, i.e. the file v1 is consumed by module v2;

iii) if $e \in Ef' \subset (F \times F)$ then the edge represents the casuality relationship between file v1 and v2 as described in section 10.3. □

DEFINITION 2.

A pair of nodes v and w in a configuration is said to be DIRECTLY EQUIVALENT, denoted by $v \langle\approx\rangle w$, if and only if:

i) v=w; or
ii) there is such a SYNONYM statement S:(u1,u2,...uk) that for some

i, j, v=ui, w=uj. □

We denote by ≈ the transitive closure of < ≈>. We will say that v and w are SYNONYMOUSLY EQUIVALENT if and only if v ≈ w.

Clearly, ≈ is an equivalence relation, defining the partition of Vf' into equivalence classes Vf'/ ≈. It can also be extended to equivalence among edges in Ef' by defining:

<v1,v2> ≈ <v3,v4> iff v1 ≈ v3 and v2 ≈ v4.

Hence the projection P: Vf' -> Vf'/ ≈ transforms Ef' into Ef'/ ≈ and also the DIRECT CONFIGURATION GRAPH Gf' into the CONFIGURATION GRAPH Gf defined as follows.

DEFINITION 3.

A CONFIGURATION GRAPH is a graph Gf=(Vf,Ef) such that Vf=Vf'/ ≈ and Ef=Ef'/ ≈. □

We will denote by Nf the number of nodes in Gf and by Ef the number of edges.

11.4.3 CONSTRUCTION OF THE CONFIGURATION GRAPH

The construction of the configuration graph is performed during syntax analysis stage.

To illustrate the composition process, we select the EBNF productions of the path statement and a sample CSL specification containing two path statements:

```
M: MX -> F: TEST1;
M: M1 -> F: TEST1 -> M: M2 -> F: TEST2, TEST3;
```

We first show below a segment of the SAP program corresponding to the productions of the path statement. The graph construction mechanism is demonstrated through description of functions of various

sub-routines in the SAP program.

The following segment is a simplified EBNF description of the path statement.

```
<PATHSTMT>   ::= <M_F>/E(4)/ : /E(2)/ <MF_NAMES> [ <ARROW>
                  <M_F>/E(4)/ : /E(2)/ <MF_NAMES> /STMF/ ]*
                 /E(1)/;/CLRALL/
<ARROW>      ::= /ARROW/
<M_F>        ::= <M> | <F>
<M>          ::= /RECM/
<F>          ::= /RECF/
<MF_NAMES>   ::= <MF_NAME> /ADDNAME/ [,<MF_NAME> /ADDNAME/]*
```

FIGURE 19. Segment EBNF/WSC of the Path Statement

Following is the segment SAP program for PATHSTMT:

```
PATHSTMT: PROCEDURE RETURNS(BIT(1));
CALL $MARK;
IF M_F( ) THEN DO;
   CALL E(4); CALL LEX;
   IF LEXBUFF = ':' THEN DO;
      CALL  LEXENAB; CALL $POPF;
      IF MF_NAMES( ) THEN DO;
$SYS_002: ;
         IF ARROW( ) THEN DO;
            IF M_F( ) THEN DO;
               CALL E(4); CALL LEX;
               IF LEXBUFF = ':' THEN DO;
                  CALL  LEXENAB; CALL $POPF; CALL E(2);
                  IF MF_NAMES( ) THEN DO;
                     CALL STMF;
                     GO TO $SYS_002;
                  END;
                  ELSE DO; CALL $SUCCES; RETURN('1'B); END;
               END; ELSE DO; CALL $FAIL; RETURN('1'B); END;
            END; ELSE DO; CALL $SUCCES; RETURN('1'B); END;
         END;ELSE;
         CALL E(1); CALL LEX;
         IF LEXBUFF = ';' THEN DO;
            CALL  LEXENAB; CALL $POPF; CALL CLRALL;
            CALL $SUCCES; RETURN('1'B);
         END; ELSE DO; CALL $FAIL; RETURN('1'B); END;
      END; ELSE DO; CALL $SUCCES; RETURN('1'B); END;
   END; ELSE DO; CALL $FAIL; RETURN('1'B); END;
END; ELSE DO; CALL $FAIL; RETURN('0'B); END;
END PATHSTMT;

M_F: PROCEDURE RETURNS(BIT(1));
CALL $MARK;
IF RECM( )
THEN DO;
  CALL $POPF; CALL $SUCCES; RETURN('1'B);
END; ELSE DO;
 IF RECF( )
 THEN DO;
   CALL $POPF; CALL $SUCCES; RETURN('1'B);
 END; ELSE DO; CALL $FAIL; RETURN('0'B); END;
END;
END M_F;
```

(to be continued)

```
MF_NAMES: PROCEDURE RETURNS(BIT(1));
CALL $MARK;
IF MF_NAME( )
THEN DO;
  CALL ADDNAME;
$SYS_003: ;
  CALL LEX;
  IF LEXBUFF = ',' THEN DO;
     CALL  LEXENAB;
     IF MF_NAME( ) THEN DO;
        CALL ADDNAME;
GO TO $SYS_003;
     END; ELSE DO; CALL $SUCCES; RETURN('1'B); END;
  END;ELSE;
  CALL $SUCCES; RETURN('1'B);
END; ELSE DO; CALL $FAIL; RETURN('0'B); END;
END MF_NAMES;
```

FIGURE 20. Segment SAP of the Path Statement

E(1), E(2) and E(4) are the error message routines. They report missing ';', unrecognized node name(lexical analysis error) and missing ':' respectively.

$MARK is a routine that pushes a string of blank characters into the error stack. Routine $SUCCESS empties the error code stack after parsing a production successfully. Routine $FAIL reports syntax errors. $POPF is a routine that pops an error code off the error stack after an error code related non-terminal has been parsed successfully. For instance, if the call RECM is successfully parsed, $POPF will be called(see above).

LEX is the lexical analyzer. It produces a globally accessible buffer, LEXBUFF, as output. The buffer contains a recognized symbol. LEXENAB is a routine that recognizes the symbol next to the currently recognized one. It is used when user programmed "look ahead" is needed. In fact, it is a lexical analyzer (LEX) in the form of a function.

The graph construction is carried out by i) building a local graph for each CSL statement, and ii) concatenating the local graph with a global graph. Initially, the global and local graphs are both null.

Being a recursive descendent parser, SAP scans generally only the current symbol and is not able to look ahead. Thus the semantic routines have to store information that they need to keep track of the previously parsed symbols.

A typical example of a specification in EBNF is the design of routines RECM and RECF.

Routines /RECM/ and /RECF/ perform the following tasks:

i) recognizing the key words F or M,
ii) comparing the keywords in the previous and current nodes and reporting an error if a M->M edge is found in the specification, and
iii) storing the currently scanned key word for the next comparison.

Non-terminal MF_NAME contains a routine (not shown above) that searches a symbol table for node type. If the node name has been found in the table, the corresponding node in the graph is located and the current keyword (M or F) is checked against the information stored in the table. Error message will be issued if a difference is found. Otherwise a new node with a specified keyword is created.

The function of routine ADDNAME is to create a list of MODULE or FILE names by concatenating the currently processed node to the list. The header of the list is globally accessible.

Routine STMF takes, as input, two lists of names constructed from the two sides of a "->" symbol. It then creates an edge from each node on the left-hand-side of the "->" symbol to each one on the right-hand-side.

Routine CLRALL simply concatenates the local graph with the growing global graph.

Now suppose that we have the two CSL path statements shown before. Initially the global and local graphs are both null. When SAP comes to call PATHSTMT, it then calls routine M_F. Since the

current symbol is "M", routine M_F returns "true" to PATHSTMT. PATHSTMT then calls E(4) which stacks error code "4" onto the stack. E(4) produces a message reporting missing ":" in a CSL specification. The SAP then calls LEX to get another symbol, and checks if it is ":". If ":" is the current scanned symbol, SAP calls LEXENAB to pop the code "4" off the stack; otherwise it calls $FAIL to report the missing ":" error.

In the chosen example, ":" does appear, so MF_NAMES is the next routine to be called in SAP.

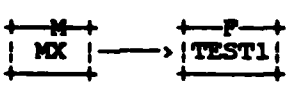
Inside MF_NAMES, routine ADDNAME is called to create a single node list. Since "," does not appear on the left-hand-side of the "->" symbol, routine MF_NAMES simply calls $SUCCES and returns to SAP.

The SAP then calls ARROW which is a routine for recognizing the "->" symbol. In the chosen CSL example, ARROW must succeed. The right-hand-side of the "->" symbol is then processed in the same way until the SAP reaches STMP which creates a local graph from both sides of the "->" symbol. The snap-shot picture at this moment is shown below.

Global Graph

(null)

Local Graph

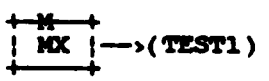


Similar to processing ":", the ";" is checked. Finally the SAP reaches CLRALL which concatenates the local graph to the global graph and sets the local graph to be null.

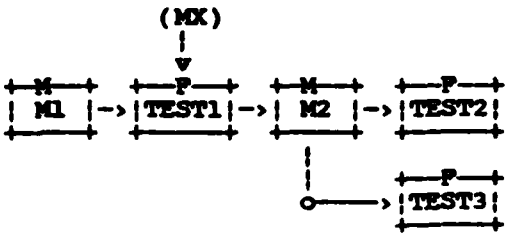
During parsing the second line, TEST1 is identified as existing, so the local process starts with this node taken from the global graph.

Before reaching routine CLRALL, we have the following situation:

Global graph

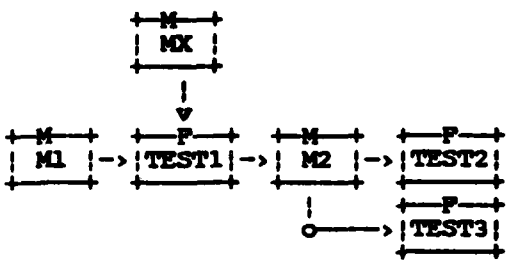


Local graph



The graphs after the completion of parsing line 2 are:

Global Graph



Local Graph

(null)

The mechanism used to process the SYNONYM statements is similar. It locates the nodes specified in a synonym statement, takes them from the global graph, and attaches them to the last node (still in the global graph) in the statement.

11.4.3.1 DATA STRUCTURES USED IN THE GRAPH CONSTRUCTION

The data structures used in graph construction is shown as follows.

```
DCL 1 NODE BASED(NPTR),
      2 TYPE    CHAR(1),
      2 DFNUMBER FIXED BIN, /* FOR MSCC FINDING */
      2 LOWLINK  FIXED BIN,
      2 STATUS CHAR(1),      /* + */
      2 M_M_LKS PTR,         /* POINTS TO STMTNO LIST */
      2 SYNHEADER PTR,      /* TO THE HEAD OF SYNONYM LIST */
      2 NAME    CHAR(10),
```

(to be continued)

```
      2 PNAME CHAR(10),
      2 LOC    CHAR(10),
      2 DEVICE CHAR(10),
      2 DIRECTORY CHAR(20),
      2 SUFIX     CHAR(3),
      2 VERSION   FIXED BIN,
      2 ORG       CHAR(4),
      2 REC_SIZE  FIXED BIN,
      2 SYN_LIS   PTR,          /* POINTS TO NEXT SYNONYMOUS NODE */
      2 PRE_LIST  PTR,          /* POINTS TO A LK LIST */
      2 SUC_LIST  PTR,          /* POINTS TO A LK LIST */
      2 W1        BIT(1),       /* FOR PREFIXED NAMES */
      2 NEW       BIT(1),       /* FOR TRAVERSING */
      2 NEXT      PTR;     /* POINTS TO NEXT NODE IN SYMBOL TABLE */

DCL 1 LK_LIST BASED(LK_PTR),
      2 CLAIMED_ORG CHAR(4), /* UNUSED */
      2 STMT_NO    FIXED BIN,/* USED TO PRINT X-REFERENCE REPORT */
      2 PT   PTR,             /* POINTS TO A F OR A M IN NODE */
      2 NEXT PTR;             /* POINTS TO THE NEXT LK STRUCTURE */
```

Gf is constructed through fields "PRE_LIST", "SUC_LIST", "SYN_LIS" and "SYN_HEAD". Field "SYN_HEAD" points to the head of a group of synonymous nodes. Every group of synonymous names is connected through field "SYN_LIS" which points to a next synonymous NODE structure.

Also note that all the modules and files in a configuration are stored in two alphabetically sorted linked lists through the pointer NEXT in structure NODE. One such list consists of all the (unique) module names and the other all the (unique) file names. The two lists are pointed by two global pointers M_HD and F_HD respectively.

11.5 COMPLETENESS ANALYSIS (PROCEDURE NAME: CMPANA)

After the configuration graph construction, a completeness analysis is performed.

•

DEFINITION 4.

A configuration graph Gf is said to be complete if and only if

i) no node is isolated, and

ii) each MAIL or POST typed FILE node in Vf has at least one producer

and consumer. □

If a FILE node has no producers and consumers, the file does not participate in the system and we conclude that the CSL specification is incomplete. If a MODULE node does not produce and consume any files, the module does not communicate with anything and is not part of the system, we also conclude that the system is incomplete.

The MAIL and POST files are supposed to communicate among modules. Lack of either a producer or a consumer indicates an incomplete definition of the system.

The analysis is a simple one-pass scan through all nodes counting the number of predecessors and successors of each node. Error messages are issued if the Gf is found to be incomplete.

Other checks, such as on "+" modules and the checks on distributivity, according to the rules stated in section 10.6.1, are also performed in the same procedure.

11.6 SEQUENCE CHECKING (PROCEDURE NAME: SEQCK)

11.6.1 PRELIMINARIES

11.6.1.1 REQUIREMENT

A dataflow analysis approach is employed to check the execution sequence of a configuration to check for conflicts in scheduling of nodes.

The limited information provided in a CSL specification, is due to the assumption that each module is ATOMIC.

DEFINITION 5.

An ATOMIC module is a module which acquires all of its input files at beginning of its execution and releases all of its output

files at the end of its execution. □

The user must assume that all the module nodes in the configuration are atomic. In many cases this may be an overly conservative assumption. In particular this becomes apparent in the case of GRP nodes as illustrated below. Further, the use of non-atomic node may cause loss of ability to conduct some checks of correctness of a configuration and also loss of some efficiency in its scheduling. The alternative for the user is to decompose non-atomic modules into atomic ones.

For instance, if

```
F:  F1 (TYP:MAIL) -> M:  MG (TYP:GRP) -> F:  F1;
```

The GRP type module MG may have two different sub-configurations each containing modules MG1 and MG2. They, however, lead to totally different configuration graphs. Note that what appears as a cycle in the above statement is in fact not a cycle in Figure 21(a), while there is a cycle containing SAM file in Figure 21(b) which is not detectable from the above statement.



FIGURE 21. Two possible sub-configurations of a GRP module

A module producing only ISAM file(s) does not impose any execution sequence constraints, thus is not required to be atomic.

11.6.1.2 TEMPORAL RELATIONS

First, we define temporal relations that can be derived from a configuration specification.

In any timing of execution of a system described by the given configuration, each module Mi is represented by a pair of time points ‹Mis,Mie›, standing for starting and ending times of its execution, respectively. Values of such starting and ending times are highly inter-dependent due to existence of certain temporal relation between execution times of modules. For our purpose it is sufficient to consider the following three temporal relations.

DEFINITION 6.

For any given pair of time intervals: M1=‹M1s,M1e› and M2=‹M2s,M2e› we say that:

M1 => M2, iff M1e < M2s

M1 -> M2, iff M2s ≤ M1s & M2e ≥ M1e

M1 || M2, iff M1s = M2s & M1e = M2e

We refer to the temporal relation "=>" as a sequential one, to "->" as a mail relation, and to "||" as a parallel one. By "<=" and "<-", we denote inverse of sequential relation and mail relation respectively. Following is a transitivity table for the defined temporal relations:

r1 \ r2	=>	<=	->	<-	||
=>	=>	no info	=>	no info	=>
<=	no info	<=	no info	<=	<=
->	=>	no info	->	no info	->
<-	no info	<=	no info	<-	<-
||	=>	<=	->	<-	||

Table 5. Transitivity table of relation r holding between M1 and M3 assuming that M1 r1 M2 and M2 r2 M3 hold.

For example, if (M1 => M2) and (M2 || M3) hold, then we can easily show that M1 => M3 as follows:

(M2s > M1e) from definition of =>
(M2s = M3s) from definition of ||
so (M3s > M1e), hence M3 => M1. □

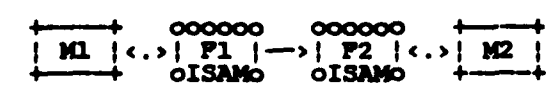
Other entries in the transitivity table can be proved similarly.

Sequential relation holds between modules which have to be executed one after another. This relation exists between the producer and consumer of a SAM file or a sequentially accessed ISAM file. A mail relation exists between a producer and a consumer module of messages through a mail or post file. It indicates that the consumer may start and before the producer and finish after the producer. Parallel relation exists between modules exchanging messages. Note that M1->M2 and M2->M1 implies M1||M2.

There are five basic connections in a configuration graph (Gf).

i) 

implies M1 => M2

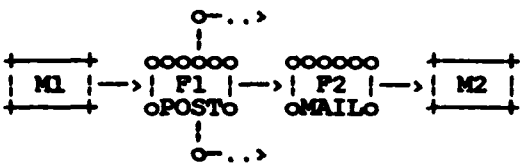
ii) 

implies M1 => M2

Where <.> means that the edges can be in either or both directions.

iii)


implies M1 -> M2

iv)


implies M1 -> M2

v)


implies no temporal relation.

To compute the temporal relations among all the module nodes in a configuration, we do the following.

I. Initially we derive all the temporal relations assumed by the basic connections stated above. Note that the derivation includes the case when a module is defined as manually initiated (i.e. prefixed by "+").

Additionally, we assume that for every module Mi, Mi||Mi.

II. Next, all consequences of constraints imposed in I are computed. Note that whenever we find Mi->Mj and Mj->Mi, we derive Mi||Mj.

The parallel relation is an equivalence. Clearly it is transitive, symmetric and reflexive. Thus we can consider an image of the sequential and mail relation under the projection mapping a set of modules into a set of module equivalence classes under ||. It is easy to see that the equivalence classes are groups of modules enclosed in MSCCs connected only by post and mail files. We will later refer to the equivalence class containing module Mi as to the component of Mi.

The derivation of temporal relations is used to perform sequence and consistency checking.

11.6.2 SEQUENCING ANALYSIS

The aim of the analysis is to locate modules that have an

inconsistent activation sequence due to sequential cycles in an MSCC in Gf.

Recall that cycles resulting from circular communication of records are detected by the MODEL compiler based on external dependency statements.

Discussed below are two forms of inconsistency and their detection.

11.6.2.1 CIRCULAR FORM INCONSISTENCY

A circular form inconsistency occurs as the result of inconsistent accessing of sequential files, or sequential access of ISAM files.

DEFINITION 7.

A circular form inconsistency is indicated by a module in a configuration whose execution may precede only after its own termination. □

It is not difficult to see that a configuration containing circular form inconsistency if and only if there exists some module Mi in the configuration, such that Mi=›Mi can be derived from some initial temporal relations in a configuration.

In order to efficiently detect all possible circular form inconsistencies in a configuration, we introduce another graph representation.

DEFINITION 8.

A COMPONENT GRAPH Gc=(Vc,Ec) is a graph constructed from Gf satisfying: Vc = Vf/||, and Ec = Ef/||. □

Again, it is not difficult to see that a Gf contains any circular

form inconsistency, if and only if there exists a cycle in Vc.

As mentioned, every component in Gc is an equivalent class, which corresponds to a MSCC consisting modules strongly connected by POST and MAIL (file) edges. The following algorithm finds such MSCCs in Gf.

Algorithm 1. Vc construction (Procedure name: BVC)

```
Input : Gf
Output: Vc

1. Let COUNT=1, STACK=empty and TYP(x) be the
   TYPE attribute for a node x.
2. Set all the nodes in Gf to be "new".
3. Do the following while there is a "new" node x in Gf:
4.    Call search(x).
5. End.  □

Search: Proc(ND) recursive;

11. Set ND to be "old".
12. Set ND.dfnumber=COUNT.
13. Set ND.lowlink=ND.dfnumber.
14. PUSH(ND) onto STACK.
15. COUNT=COUNT+1.
16. Do the following for every successor z of ND.
17.    If TYP(ND) ≠ "ISAM" & TYP(ND)≠ "SAM"
18.    then do;
19.      If z is "new"
20.      then do;
21.        Call search(z).
22.        ND.lowlink=min(ND.lowlink,z.lowlink).
23.      End.
24.      Else do;
25.        If z.dfnumber < ND.dfnumber & INSTACK(z)
26.        then  ND.lowlink=min(z.dfnumber,ND.lowlink).
27.       End.
28.    End.
29. End.
30. If ND.lowlink=ND.dfnumber
31. then do;
32.   Do the following until P=ND.
33.      P=POP(STACK).
34.      Call ADD_COMP(P).
35.   End.
36.   Call END_COMP(ND).
37. End.   □
```

ADD_COMP is a procedure that creates a list of elements that belong to a MSCC. Procedure END_COMP marks the end of the MSCC.

This algorithm is a variation of the depth-first search algorithm. Its correctness and complexity proofs can be found in

[Aho,74]. The complexity is MAX(Nf,Ef). Line 17 ensures that the traversing on Gf is conducted only on connections implied by -> relations (through POST and MAIL files).

Algorithm 2. Construction of Ec. (Procedure Name: BEC)

```
Input : Vc, Gf
Output: Gc

1. Do the following for every component node P (of Vc).
2.    Do the following for every element e in P.
3.       If successor(e) is a file node
4.       then do;
5.         If TYP(successor(e))=SAM or MAIL or POST
              and successor(2)(e) ≠ nil
6.         then find the component node SC which contains
7.              successor(2)(e) as a member and make
                an edge from C to SC.
8.         If TYP(successor(e))=ISAM and TYP(successor(2)(e))=ISAM
9.         then do;
10.          if successor(3)(e) ≠ nil
11.          then find the component node SC which contains
12.               successor(3)(e) as a member and make
                  an edge from C to SC.
13.          if predecessor x of successor(2)(e) ≠ nil
14.          then find the component node SC which contains x
15.               as a member and make an edge from C to SC.
16.        End.
17.      End.
18.   End.
19.End. □
```

Note that line 13 of the algorithm tests if a target node of an ISAM F-F edge has any predecessor, if it does, a new edge in Gc is created (see definition of basic connections and definition of Gc).

Algorithm 2 has complexity O(Nc*Nc), because every node in Gf may be connected to every other nodes in Gf.

The data structures for constructing Gc is as follows:

```
DCL 1 COMP_NODE BASED(C_PTR),
      2 COMP_NO    FIXED BIN, /* COMPONENT ID */
      2 NEW        BIT(1),    /* FOR MSCC FINDING */
      2 DFNUMBER   FIXED BIN, /* FOR MSCC FINDING */
      2 LOWLINK    FIXED BIN, /* FOR MSCC FINDING */
      2 SCDED      BIT(1),    /* MARK=1 IF SCHEDULED */
      2 PRES       FIXED BIN, /* NUMBER OF PREDECESSORS */
      2 ELE_LIST   PTR,       /* POINTS TO A C_LIST */
      2 SUC_LIST PTR,         /* POINTS TO A LK_LIST */
      2 MAX_D      FIXED BIN, /* DIAMETER */
      2 NEXT       PTR;       /* TO NEXT COMP_NODE */

DCL 1 C_LIST BASED(CL_PTR),
      2 ROOT_MK  BIT(1),      /* FOR MSCC FINDING */
      2 ND_POINT PTR,         /* POINTS TO A NODE STRUCTURE */
```

```
2 NEXT     PTR;        /* POINTS TO NEXT C_LIST */
```

The list of all components in Gc is pointed by a global pointer COMP_HD.

After the construction of Gc, the error detection routine (SEQERR) first finds all the MSCC in Gc using the same algorithm as algorithm 1 without line 17 and then use the following algorithm to report error.

Algorithm 3. Report Sequencing Error.
(Procedure Name: RPTERR).

Input : List of MSCCs in Gc.
Output: Error message, if any.

1. Do the following for every member x in the list of MSCCs.
2. If x contains more than one element then ERROR (SEQ1).
3. If x has an edge in Gc which leads to itself then ERROR (SEQ2).
4. End. □

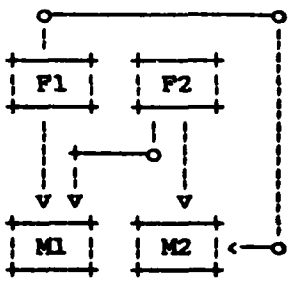
The error messages are shown as follows:

```
(SEQ1) AN INCONSISTENT MULTI-NODE MSCC FOUND IN CONFIGURATION
       CONSISTS OF:
   - element 1
   - element 2
       :
       .
(SEQ2) AN INCONSISTENT COMPONENT FOUND IN CONFIGURATION
       CONSISTS OF:
   - element 1
   - element 2
```

Note that if no error is detected, Vc is used in diameter finding calculation (section 11.8).

11.6.2.2 NON-CIRCULAR FORM INCONSISTENCY

Non-circular form inconsistency can be depicted as follows.



F1 and F2 are sequential files. Both M1 and M2 consume files F1 and F2. It is necessary to schedule M1 and M2 sequentially.

We will refer to the problem as SPS (Sequential File Sharing).

A solution is to impose a sequential relation between M1 and M2. The following algorithm identifies the problem and resolves it if it is possible.

Algorithm 4. Solving SPS problems in Gf. (Procedure name: SLVSPS)

Input : Gf and Gc
Output: A modified Gc, if a solution is found

Data structure used:
A queue Q of component identifiers.

Let Si be a list of files with type="SAM" and device="TAPE" which are consumed by module Mi.

Let PRi be a list of component identifiers preceeding a component Ci.

Functions used:
PUTQ(e) is a procedure which puts e to be on the end of Q and
TAKEQ( ) is a function which returns the first element in Q.

```
 1. Set COUNT=0.
 2. For all module node Mi in Vf do the following:
 3.      Set S = ∅
 4.      For all predecessor v of Mi with device="TAPE" do:
 5.          S = S  ∩ (v).
 6.      End.
 7.      If S contains more than one element
         than do;
 8.        COUNT=COUNT+1.
 9.        Set  = S.
10.      End.
11. End.
12. If COUNT>1
13. then do;
14.    Set Q=empty.
15.    For all Ci in Vc do the following:
16.        If Ci has no predecessors then PUTQ(Ci).
17.        Set PRi = ∅.
18.        Set Ci = "new".
```

```
19.    End.
20.    Do while Q is not empty:
21.       Ci=TAKEQ( ).
22.       For all successors Cj of Ci do:
23.           PRj = PRj U PRi U (cj).
24.           If Cj is "new" then PUTQ(Cj).
25.           Set Cj = "old".
26.       End.
27.    End.
28.    For i=1 to COUNT do:
29.        For j=i+1 to COUNT do:
30.            If Si ∩ Sj has more than one element then do:
31.               Let Cl be the component containing Si and
                      Ck be the component containing Sj.
32.               If Cl=Ck then report ERROR (SPS1).
33.               If Cl ∉ PRk & Ck ∉ PRl
34.               then do;
35.                 Add an edge from Cl to Ck in Gc.
36.                 Call CORR(k,l).
37.               End.
38.            End.
39.        End.
40.    End.
41. End. □

CORR (k,l) recursive:
   PRk = PRk U PRl.
   For all successors Cx of Ck call CORR(x,l).
End.  ⊟
```

This algorithm first finds for each module a set of "TAPE" predecessors (lines 1-8). If variable COUNT is greater than one, it implies possibility of existence of SPS problem in Gf.

Lines 14-27 compute the transitive closure of "=›" relation along every path in Gc and associate the computed results with each component. In other words, for all components in Gc, if Ci=›Cj then i ∈ PRj.

Lines 28-38 compute pairwise intersection among all the module nodes that have Si containing more than one element. If the intersection of Si and Sj contains more than one element, the two modules Mi and Mj have to be in a sequential relation. If Mi and Mj are in the same component, no sequential relation can be imposed (see definition of Gc); an error is reported. Line 33 tests if Mi is sequentially related to Mj in either direction. If they are not related, a new edge is added in Gc to impose a "=›" relation among Mi and Mj, and transitive closure of newly added relation is computed by procedure CORR.

Note that the sequence of imposing sequential relation is not important in finding the solution for a SFS problem.

DEFINITION 9.

A solution for a SFS problem is a partially ordered set Q of N modules sharing M sequential tape files, where N>1 and M>1. And Q is compatible with Gc - the Gc including Q is acyclic. □

Proposition 1: If Algorithm 4 does not report error, then either there is no SFS problem in Gf or it has found a solution of the SFS'.

Proof.

If Algorithm 4 does not report an error, this implies:

i) The condition in line 32 is falsified
   —› If a Q is found, it is compatible with Gc.
ii) There are only two cases existing from lines 33-38:
   a) The condition in line 33 is falsified
      —› there exists a sequential relation between the Cl and Ck in Gc.
   b) The condition in line 33 is true
      —› lines 35 imposes a sequential relation from Cl to Ck.
   —› If the condition in line 32 is not true -› A Q is found. □

Proposition 2: If there exists a solution for a SFS problem, then Algorithm finds it.

Proof.

If a solution exists, then Q exists and it is compatible with Gc.

Suppose Algorithm 4 does not find Q. There is only one possibility:

line 32 reports an error.

The condition in line 32 is "Cl=Ck"

—› there exists two elements Cl,Ck in Q, such that Cl||Ck.

—› If the condition is wrongly programmed, then lines 33-38 are executed. From Proposition 1, either Cl=›Ck or Ck=›Cl will be imposed. In conjunction with Cl||Ck, by the definition of Gc, there must be a cycle in the Gc including Q - contrary to the assumption.

—› If the condition is correctly programmed, then Q is not compatible with Gc - contrary to the assumption. □

The complexity of the algorithm can be analyzed as follows. Lines 1-8 take NM*(Nf-NM) steps to compute Si's, where NM is the number of module nodes in Gf.

We assume that the graph Gc is cycle free. Therefore lines 10-21 take Nc+Ec steps to compute PRi's.

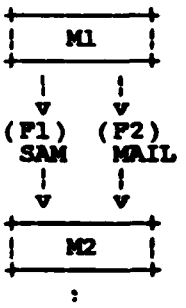
Finally lines 23-35 takes at most Nc*(n*(n-1)/2) steps to check all the possible pairs of modules, where n denotes the number of modules consuming more than one tape file.

The total complexity of the algorithm is therefore O(Nc*n*n).

11.7 SCHEDULING (PROCEDURE NAME: SCHEDULE)

11.7.1 MORE CONSISTENCY ANALYSIS

A configuration that has passed the sequence checking may still cause some problems at runtime. For example, the following configuration is inconsistent in that we can derive two conflicting temporal relations from it: M1=›M2 and M1-›M2.



DEFINITION 10.

A configuration is inconsistent, iff there exist two modules Mi, Mj in Gf, such that Mi=›Mj can be derived by one path and Mi->Mj or Mj->Mi, by some other(s). □

Note that the inconsistency definition includes circular form inconsistency.

Using the similar idea for inconsistency detection, the above definition calls for a new extended parallel relation ||| which holds among every pair of modules Mi and Mj either Mi->Mj holds or Mj->Mi holds. Evidently ||| is also an equivalence relation and the equivalent classes under ||| properly include the equivalent classes of ||.

To verify the consistency of a configuration, we define the following graph.

DEFINITION 11.

An EXTENDED COMPONENT GRAPH Ge=(Ve,Ee) is a graph whose sets of nodes Ve and edges Ee are defined as follows:

Ve= Vf/|||

Ee= "=›"/|||, or in other words, ‹C1,C2› ϵ Ee, iff there are such modules M1 ϵ C1 and M2 ϵ C2 that M1 =› M2. □

We will denote the number of nodes in Ge by NDe and number of

edges in Ge by EGe. Now it is also not difficult to see that if Ge is cycle free, then the execution of a configuration is feasible.

The detection of an inconsistent configuration is accomplished by first constructing Ge and then finding MSCC in Ge (using the same algorithm as for Gc construction). Clearly, all the module nodes in an extended component can be initiated at the same time by the definition of -> relation.

11.7.2 SCHEDULING

To achieve maximum concurrency, we start execution of each module as early as possible. The only delay on initialization is due to the sequential relations existing between modules.

By the definition of Ge, we can see that Ge can be used directly as the schedule graph.

Following are the algorithms for Ge construction.

Algorithm 5. Construction of Ve. (Procedure name: BVE)

```
Input : Vc, Gf
Output: Ve (also referred as a set of extended component nodes).

1. Set all the Mi nodes in Gf to be "new".
2. Set all the edges in Gf to be "new".
3. Do the following for every member Qi in Vc which contains
     "new" nodes.
4.   Allocate a component node P for Qi and make all the module nodes
     in Qi be the element in P.
5.   Do the following for every element e in P.
6.      Mark e "old".
7.      Do the following for every predecessor or successor x of e
           which is "new" and connected to e by a "new" edge.
8.         Mark x "old".
9.         Mark the edges e->x and x->e "old".
10.        If TYP(x)=MAIL or TYP(x)=POST
11.        then make x the new element in P
12.     End.
13.  End.
14.End. □
```

Algorithm 6. Construction of Ee. (Procedure name: BEE)

```
Input : Ve, Gf
Output: Ge

1. Do the following for every component node P (of Ve).
2.    Do the following for every element e in P.
```

```
3.        If successor(e) is a file node
4.        then do;
5.          If TYP(successor(e))=SAM and successor(2)(e) ≠ nil
6.          then find the component node SC which contains
7.               successor(2)(e) as a member and make
                 an edge from C to SC.
8.          If TYP(successor(e))=ISAM and
               TYP(successor(2)(e))=ISAM
9.          then do;
10.           if successor(3)(e) ≠ nil
11.           then find the component node SC which contains
12.                successor(3)(e) as a member and make
                   an edge from C to SC.
13.           if predecessor x of successor(2)(e) ≠ nil
14.           then find the component node SC which contains x
15.                as a member and make an edge from C to SC.
16.         End.
17.       End.
18.   End.
19.End. □
```

In constructing Ve, in the worst case, the algorithm traces every node and edge in Gf exactly once. Thus the complexity of the algorithm is O(Nf+Ef).

In constructing Ee, the function successor(k)(e) returns the k-th successor of e. In line 7, the algorithm creates edges derived from SAM file connections in Gf. Lines 10-16 create edges implied by ISAM F-F causality relations.

The complexity of construction of Ee is O(NDe*NDe).

The data structures for Ge construction is the same as for Gf provided the field M_M_LKS is used to point all the elements in an extended component. The list of Ve is pointed by the global pointer: ECOMP_HD.

The following algorithm reports inconsistency error.

Algorithm 7. Report Inconsistency Error.
(Procedure Name: CSTERR).

Input : List of MSCCs in Ge.
Output: Error message, if any.

```
1. Do the following for every member x in the list of MSCCs.
2.    If x contains more than one element then ERROR (SCD1).
3.    If x has an edge in Gc which leads to itself then ERROR (SCD2).
4. End. □
```

Detailed error messages are shown below.

```
(SCD1) CYCLES FOUND IN A SCHEDULE GRAPH CONSISTING OF:
       -PAR NODE: element 1
       -PAR NODE: element 2
                  :
                  .
(SCD2) A SIMPLE CYCLE IS FOUND IN A SCHEDULE GRAPH CONSISTING OF:
       -ELE: element 1
       -ELE: element 2
                 :
                 .
```

11.8 MSCC DIAMETER EVALUATION (PROCEDURE NAME: FDMAX)

The diameter of a MSCC in the configuration graph is used by the termination control algorithm for terminating iterative solutions of distributed simultaneous equations (described in Part III).

DEFINITION 12.

Let p(Mi,Mj) be the number of module nodes in the shortest path from a module node Mi to a module node Mj, and N be the number of module nodes in a MSCC. The DIAMETER of a MSCC is equal to

MAX(ML(M1),ML(M2),...,ML(MN)) - 1

where ML(Mi) is defined as

MAX(p(Mi,M1),p(Mi,M2),...,p(Mi,MN)). □

In the current implementation, the Configurator computes the diameter for each MSCC found in a configuration graph and passes this value to each individual module through a logical name assignment JCL command:

$DEFINE MAX_D "diameter".

The individual modules can then get the diameter through the logical name MAX_D.

To compute the diameter of a graph, we need to find the lengths of the shortest paths between any two nodes in the MSCC. There have

been a number of algorithms proposed for performing this task, usually with complexity O(N**3) [Berztiss, 71]. Since we expect that in majority of cases, a MSCC has the number of edges proportional to N, rather than N**2, we selected the following algorithm with complexity O(N*E).

Algorithm 8. Finding diameter of a MSCC. (Procedure Name: FDMAX)

```
Input : A MSCC (from algorithm 1)
        represented as a set of multi-linked lists
Output: Diameter (MAX) of the MSCC

Data structure used:
   A queue Q with a pair of integers as elements: (e,d),
   where is the identifier of a node and d is the distance
   value.

Let TAKEQ( ) be the function returning the pair of first element in Q,
    and PUTQ(e,d) be the function that puts e at the end of Q, d is
    interpreted as the distance of e from the root.
1.  MAX=0.
2.  Do the following for each module node Mi in the MSCC:
3.     empty Q.
4.     set all elements in the MSCC to be "new".
5.     PUTQ(Mi,0).
6.     Do the following while(Q is not empty).
7.        Let (e,d)=TAKEQ( ).
8.        For each successor succ of e do.
9.           If succ is "new"
             then do;
10.            Set succ to be "old".
11.            If succ is a module node
               then do;
12.              PUTQ(succ,d+1).
13.              If d  > MAX then MAX=d+1.
14.            End.
15.            Else call PUTQ(succ,d).
16.          End.
17.       End.
18.    End.
19. End.   □
```

The above algorithm correctly computes the diameter of a MSCC.

Proof.

Note that in the maximum distance calculation, the file nodes are "transparent", i.e. a path Mi->Fl->Mj is of length 1 rather than 2.

The algorithm uses a global variable MAX for storing the final result. MAX is constantly modified during the processing (line 13).

We must compute ML for every module node in the MSCC. This is

done by lines 3-19. Line 5 puts the "root" of a search tree (Mi) into Q. Lines 6-18 find ML(Mi). The search tree is constructed by using Q in the following way. The immediate successors are searched by line 8. If the successor is a module node, line 12 gets d+1 as the distance of the successor. Otherwise, the same d is pushed onto Q. The global variable MAX is modified whenever modified d exceeds it.

Lines 9 and 10 ensure that every node in the MSCC is to be visited only once in evaluation of ML(Mi), consequently every edge in the MSCC is visited exactly once during this process.

Since every node can be put in Q only once, we can conclude that the loop from lines 6 to 18 must terminate.

Now we prove the correctness of the algorithm by induction on the distance value k. To do that we first consider the following equivalence:

(1) p(Mi,Ms)=k <=> (Ms,d) has been in Q and d=k.

For k=0, we have exactly one node, such that p(Mi,Ms)=0, i.e. Ms=Mi. For this node Mi, line 5 defines a pair (Mi,d) with d=0. Conversely, suppose that there is a node Ms such that a pair (Ms,d) is in the Q and d=0. Line 12 implies that any module node different than Mi in the MSCC will have distance d at least by one greater than 0. Thus Ms=Mi.

Now suppose that the equivalence holds for all k $\leq$n. Let us consider a module node st, such that p(Mi,st)=n+1. There must be a module predecessor s of st, such that p(Mi,s)=n and st must be new when s is taken from Q. Thus under the inductive hypothesis, let (st,dst) and (s,ds) be the elements in Q, dst=ds+1=n+1. Conversely, if for some module node st, (st,dst) is an element in Q and dst=n+1, then there must exist some module node s, such that s is the predecessor of st, and (s,ds) is an element in Q, ds=dst-1=n-1=n. Hence p(Mi,s)=n and finally p(Mi,st)=p(Mi,s)+1=n+1. ∎

Furthermore, since line 13 conditionally modifies MAX whenever d is modified, it is trivial to see that MAX records the diameter of MSCC. □

The complexity of the algorithm can be shown as the following. Let N be the number of module nodes in MSCC, and E be the number of edges. It takes O(N) steps to go through the outer loop from line 2 to line 19. The number of steps to compute line 6 to line 18 is E+N because every node and every edge have to be visited precisely once. For MSCC with N>1, obviously E ≥N, thus the total complexity for the algorithm is O(N*E). □

11.9 CODE GENERATION (PROCEDURE NAME: GCODE)

The code generated by the Configurator is dependent on the available operating system. The current implementation uses VAX/VMS operating system, version 3.6. The Configurator generates a set of programs in JCL and PL/I that initiate and execute the system.

The implementation of the system described in a CSL specification consists of programs on three levels under VMS.

i) Main (or sub-main) JCL program level (one for each configuration specification).
ii) Individual JCL program level (one for each module in a configuration).
iii) Individual program level (one for each module).

A main (or sub-main) JCL program performs three tasks, namely

i) create necessary mailbox(es) by calling VMS mail server ($RUN crembx),
ii) submit individual (or sub-main) JCL programs for execution ($SUBMIT), and

iii) at the end of submission of modules for execution, the JCL program determines when all the modules executions have been completed. (This is done by using $SYNCHRONIZE commands.) This is necessary in the case that the main JCL program corresponds to a GRP type node, which possibly may have to be synchronized with other modules.

The five types of commands in an individual JCL program are arranged in the following way:

i) commands for synchronization ($SYNCHRONIZE)
ii) commands for assigning logical file names to the physical ones ($DEFINE)
iii) MSCC diameter definition (optional) ($DEFINE MAX_D)
iv) command for activating a module ($RUN)
v) commands for disconnecting the logical file assignments ($DEASSIGN)

A SUBMIT command in VMS puts a JCL program into a job queue. It also gives an external name (JCL accessible) to the job in the queue. VMS will then schedule the jobs in the job queue, according to the delay commands in respective jobs, and execute them in a simulated parallel fashion. A SYNCHRONIZE command is used to delay the execution of a command until the completion of a specified job. Commands DEFINE and DEASSIGN connect and disconnect the logical file names respectively. When a RUN command is executed, the executable code produced from the PL/I programs generated by the MODEL compiler is located and executed.

The following graph shows the execution of a configuration system.

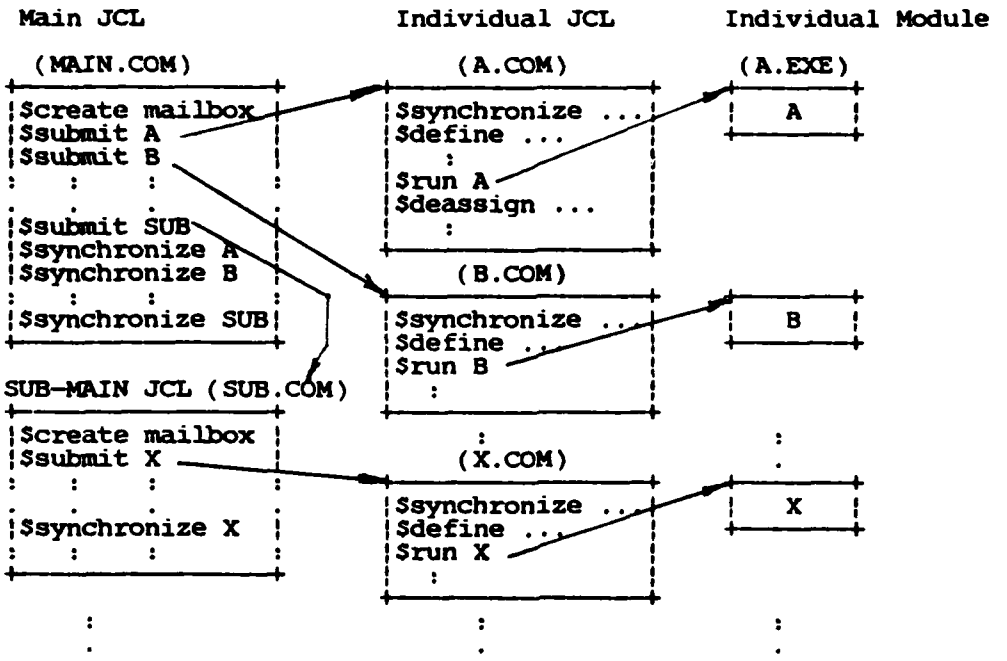


FIGURE 22. Execution of a Configuration System

The above figure shows the first two levels of a configuration system. The first level main JCL is "MAIN.COM" which submits all its constituents: A,B,... and SUB, which is a GRP node in the main configuration representing a sub-main JCL on the second level. The file "SUB.COM" is generated by a separate run of the Configurator. The "EXE" files are executable codes of all individual modules generated some high-level language compiler.

This scheme can also be used to execute programs in a computer network.

11.9.1 MAIN JCL PROGRAM GENERATION

The main JCL program is generated simply by scanning the modules in the system.

Algorithm 9. Generating main JCL programs. (Procedure name: GCODE)

Input : List of modules in Gf
Output: The main JCL program

```
1. Generate "compile", "link" and "run" commands for the
   mailbox creation program.
2. Do the following for each member e in the list.
3.    If TYP(e) ≠ "+" & TYP(e) ≠ "MAN"
4.    then generate "$SUBMIT e /NAME=e ".
5. End.
6. Do the following for each member e in the list.
7.    If TYP(e) ≠ "+" & TYP(e) ≠ "MAN"
8.    then generate "$SYNCHRONIZE e ".
9. End. □
```

The generation of the SYNCHRONIZE command is based on the sequential relations. For example, if component C1 is a predecessor of component C2 in Gc and if M1 ϵ C1 and M2 ϵ C2, then the JCL programs for M1 and M2 are:

```
(M1.com)                    (M2.com)
$RUN M1                     $SYNCHRONIZE M1
                            $RUN M2
```

Lines 5-8 are designed to ensure proper synchronization sequence for all GRP modules. It has the effect that every main (or sub-main) JCL program will not finish until each and every submitted job finishes.

A GRP module represents a sub-configuration. It implies a separate run of the Configurator which generates a sub-main JCL program. At the time of a (relative) global configuration, the details of the sub-configurations are invisible. For a module node whose sequential predecessor is a GRP module, direct synchronization with any modules in the GRP predecessor is impossible. Therefore that module has to be synchronized with the JCL program of the GRP predecessor node. Since every main (or sub-main) JCL program finishes only after the completion of all the submitted jobs, proper synchronization is thus achieved.

11.9.2 INDIVIDUAL JCL PROGRAM GENERATION

To achieve maximum concurrency, every individual JCL program synchronizes only with its direct sequential predecessor modules (in Gf).

The following is the algorithms which produces individual JCL programs.

Algorithm 10. Producing individual JCL programs.
(Procedure name: GCODE)

```
Input : Gf
Output: individual JCL programs for each module

0. Do the following for each module e in Vf.
1.    If TYP(e)≠"MAN"
2.    then do;
3.      Create a JCL file for e (named 'e'.COM).
4.      Do the following for every sequential predecessor module Mx
5.         Generate "$SYNCHRONIZE Mx/NAME=Mx"
6.      End.
7.      If TYP(e) ≠ "GRP"
8.      then do;
9.        Generate "$DEFINE logical file name"
10.       If the MSCC containing e has diameter >0
11.         then generate "$DEFINE MAC_D diameter".
12.       Generate "$RUN" and "$DEASSIGN" commands for e
13.     End.
14.     Else do;
15.       Generate "$SUBMIT Se/NAME=Se".
16.       Generate "$SYNCHRONIZE Se".
17.     End.
18.     Close 'e'.COM.
19.   End.
20.End.  □
```

Where "Se" in the algorithm is the name of "e" prefixed by an "S".

Note that although there may be a chain of "=>" relations, the synchronization among two modules can be achieved by synchronizing only the closest predecessor(s). For example, if Mi1,Mi2 => Mj => Mk, then Mk has only to be synchronized by Mj and Mj has to be synchronized by Mi1 and Mi2.

A module node of type GRP has a special JCL program generated which contains:

a) synchronization command(s),

b) a submit command for submitting a JCL program named Se (generated by a separate run of the Configurator for a sub-configuration), and

c) a synchronization command for synchronizing the termination of the JCL program with the termination of Se.

Since Se has its own termination synchronization with the modules

in the sub-configuration, a hierarchical execution pattern is then achieved.

The convention of naming CSL specifications for sub-configurations is that every sub-configuration must have a name with a leading character "S" and its representative GRP node (in the relative main configuration) must have the name without the leading "S".

The complexity of this algorithm is O(Nm), where Nm is the number of modules in a configuration.

11.9.3 PL/I PROGRAM GENERATION

A mailbox-creation PL/I program is generated by scanning all the MAIL files in the system. The existence of a mailbox file is indicated by a MAIL file node used as the source of some module node. The following algorithm traces only the source MAIL files and generates the mailbox file creation PL/I program.

Algorithm 11. Mailbox creation. (Procedure name: GCODE)

```
Input : Gf, list of files in Gf
Output: A PL/I program which creates all the necessary mailbox(es)

1. Set fst='1'b.
2. Do the following for every element e in the file list.
3.     If TYP(e)=MAIL and successor(e) is a module node
4.     then do;
5.       If fst
            then create heading of a PL/I program and declaration part.
6.       Set fst='0';
7.       Generate mailbox creation PL/I statements for e.
8.     End.
9. End.
10.If fst='0'b then generate the closing part of the PL/I program.  □
```

The use of the variable fst is to prevent the generation of empty mailbox creation programs when there is no source MAIL file in the system. The proof of the correctness of this algorithm is immediate and omitted here.

The mailbox deletion program is generated in the same way as for the creation program except that the VMS utility used is not for creation but deletion.

The complexity of this algorithm is O(Nf), where Nf stands for the number of FILE nodes in the system.

11.10 CONFIGURATION DOCUMENTATION (PROCEDURE NAME: GRPT)

As mentioned before, there are seven reports generated by the Configurator to aid the user in developing and debugging.

The CSL listing report is generated by procedure LEX.PLI during lexical analysis.

The cross reference report is generated by a procedure XREF.PLI by listing the two symbol tables alphabetically.

All the other reports are generated by a procedure GRPT.PLI.

The MSCC report is generated printing all the elements in Vc connected by the edges in Gf. This report can be used to identify possible circular form inconsistency.

The extended component graph Ge and configuration graph Gf are printed in adjacency matrix form based on Gf and Ge respectively.

The report of module-file cross reference is produced by listing all successors of any node in Gf.

The listing of JCL and PL/I program is generated only if configuration programs are desired. The user may suppress the generation of the programs and use the Configurator only for checking a system design.

The error and warning message report reports the syntax and semantic errors as well as the errors detected during completeness and consistency analysis.

All the error/warning messages produced by the Configurator are listed in Appendix C.

PART III

MODIFICATIONS TO THE MODEL COMPILER

CHAPTER 12
OBJECTIVES OF THE MODIFICATIONS

The major objective of the modification to the MODEL compiler is to enable each module to operate concurrently with other modules in an overall configuration. As explained in Part I, an overall MODEL specification may be partitioned into a number of of modules. A specification of a module consists of only variable declarations and equations. Variables in each module are also declared whether they are SOURCE or TARGET - meaning that they are in files which are external to the modules. The module then interacts with its external files. Alternatively files may be internal to the module. To have variables shared by modules, it is necessary that the variables be declared external to these modules. The implementation of the overall specification is carried out on two levels. The Configurator - described in Part II, implements the connection of modules and files to form a network. The modified MODEL compiler - discussed in this part - generates a program for computing an individual module which communicates with other modules. The old MODEL compiler documented in [Lu, 81] was developed for implementing a specification by a single sequential program without communication with other modules. Namely, the operations necessary for concurrent programming were not supported in the old MODEL compiler. The contribution of the research described in this part is to incorporate modularity and communication into the MODEL language. It can be divided into four areas as follows.

External dependencies - In composing a specification of a module, the specifier may want to have other modules provide a function which,

given values of certain arguments, returns values of some variables. To provide the arguments, the specifier must declare them to be members of record(s) of TARGET files. To refer to the result values, the specifier must declare them to be members of records of SOURCE files. It is required to declare the dependency of the result variables or the argument variable. The specifier needs not declare or ever know the names of the other modules which provide the function, or the definition of the function. This is referred to in the following as external dependency and its implementation is described in Chapter 14.

MAIL and POST files - External files may be assumed to reside on respective devices. Such files are declared as having a sequential (SAM) or indexed sequential (ISAM) organization. However, files which connect modules need not have to reside as a whole on a device but may be communicated piecewise between the concurrently computing producer and consumer modules. Such files must be declared by the specifier as having a MAIL or POST organization. MAIL and POST files are similar in function to the mailbox and post office of a postal system, respectively. The modification to the MODEL compiler to support usage of MAIL and POST files are described in chapter 15.

Concurrent update of ISAM files - ISAM files may also be used for connecting modules. In this case, the modules connected may be both producers and consumers of the file, concurrently. It is necessary therefore to provide a locking mechanism to protect the shared data, so that the connected modules do not interfere with each other. If the updating is only of a single record at a time (in the generated program), the locking mechanism is provided through the facilities of the VMS operating system. Otherwise, the user is warned to include the locking algorithm that allows exclusive access to the ISAM file in the module's specification. This is described in Chapter 16.

Inter-module simultaneous equations - When there is in the overall specification a set of n simultaneous equations with n unknowns which span more than one module, then all the respective

module programs must cooperate in the solution of these equations. In the old MODEL compiler, the Gauss-Seidel method is used when a set of simultaneous equations is confined to one module [Greenberg, 81]. The old MODEL compiler is extended for the multi-module cases. The solution of the equations in each module continues to use the Gauss-Seidel method. However, the results are communicated iteratively to the other effected modules. A Jacobi-like method is used for inter-module iterative solution. Iterations of communications and solutions continue until overall convergence conditions are attained. This extension is described in chapter 17.

The following figure shows the procedures in the MODEL compiler and their communications. As shown, the system consists of five phases: syntax analysis, array graph analysis, range and data type propagation, scheduling and code generation. The modified or added modules are marked with asterisks.

The description of the modifications to the MODEL compiler in the following chapters follow the order of the phases in the MODEL compiler. The reader who desires further detail may refer to respective chapters in [Lu, 81] for additional information. The description of each modification starts with a brief description of its objective or function, followed by description of methods, algorithms and data structures used in the phases as appropriate.

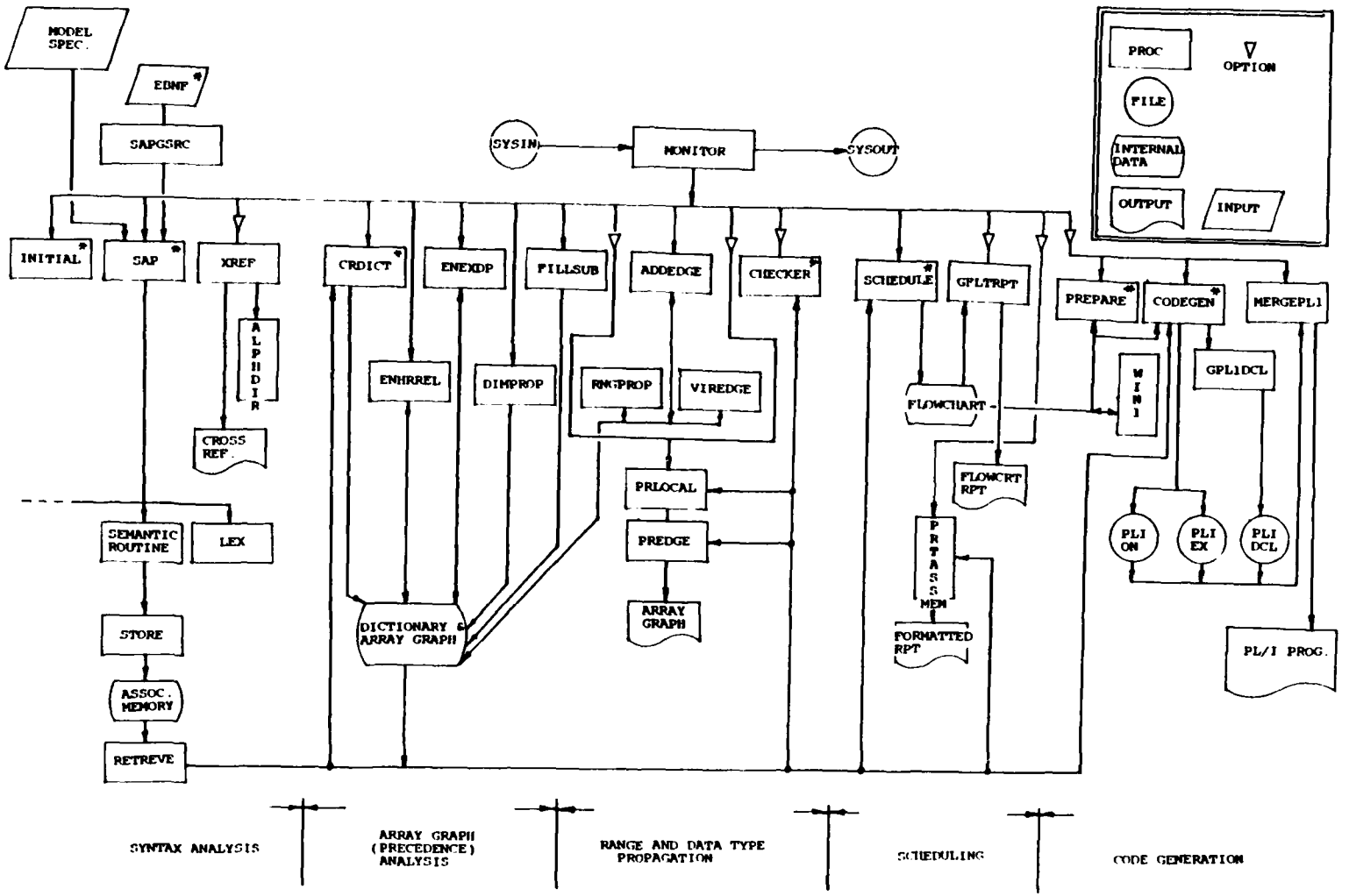


FIGURE 23. Processing stages in MODEL compiler

CHAPTER 13

EXTERNAL DEPENDENCY

13.1 FUNCTION OF EXTERNAL DEPENDENCY

In specifying a module, the user may utilize services of other modules to perform a certain function. The specified module must contain the arguments of the function in records of target files and the return results in records in source files. Additionally, to denote that there is a causal connectivity between the arguments and the results, the user must also provide an equation stating the dependency of the latter on the former. This is referred to as external dependency statement.

An external dependency consists of a pseudo function called DEPENDS_ON. It represents a dependent relationship among its left-hand-side target record(s) and right-hand-side source record(s).

The following is an example illustrating the syntax and semantics of the external dependency:

```
MODULE: M1;
SOURCE: MF;
TARGET: NF;
1 MF FILE ORG: MAIL,
  2 MR(*) RECORD,
    3 M FLD (PIC '9999');

1 NF FILE ORG: MAIL,
  2 NR(*) RECORD,
    3 N FLD (PIC '9999');
I SUBSCRIPT;
N(I) = M(I)+1;
MR(I) = DEPENDS_ON(NR(I-1));

(END.MR(I),END.NR(I))=I=1000;
```

```
MODULE: M2;
SOURCE: NF;
TARGET: MF;
1 NF FILE ORG: MAIL,
  2 NR(*) RECORD,
    3 M FLD (PIC '9999');

1 MF FILE ORG: MAIL,
  2 MR(*) RECORD,
    3 M FLD (PIC '9999');
I SUBSCRIPT;
M(I) = IF I=1 THEN 1
       ELSE N(I-1)+1;
NR(I) = DEPENDS_ON(M(I));
(END.NR(I),END.MR(I))=I=1000;
```

M1 and M2 constitute a concurrent system, connected through files MF and NF. NR and MR are two records in modules M1 and M2 respectively. The vector of records MR consists of all the odd numbers from 1 to 1999 and NR of all the even numbers up to 2000.

In module M2, M(I) is defined as being 1 for I=1 and as N(I)+1 for other values of I. Similarly in M1, N(I) is defined as being M(I)+1. The specifier of M1 has to state the external dependency among variables M and N, which is provided externally, i.e. by module M2. Similarly, the specifier of M2 has to state the external dependency among variables N and M, which is provided externally, i.e. by module M1.

13.2 SYNTAX ANALYSIS

DEPENDS_ON is a pseudo function with variable argument list. It is treated as an ordinary equation with LHS as the dependent variable and RHS as the independent variable(s). Thus there is no change to the syntax analysis phase.

13.3 ARRAY GRAPH ANALYSIS (PRECEDENCE ANALYSIS)

In building the symbol dictionary, a procedure INITIAL is called to check the attributes of each symbol. If a symbol is a built-in function, appropriate mark is made in the dictionary. Since the DEPENDS_ON statement is treated as a built-in function, it is placed in the 43rd position in the array FCNAMES by program INITIAL. The other functions are either MODEL built-in functions or PL/I built-in functions.

No change is made in array graph construction.

13.4 RANGE AND DATA TYPE PROPAGATION

The data type check routine CHECKER, which checks for number of arguments of a function, by-passes the DEPENDS_ON function. This

enables the DEPENDS_ON function to have arbitrary number of arguments with arbitrary data type. Because the data nodes of a DEPENDS_ON function must all be RECORD nodes, there should be no data types assigned.

13.5 CODE GENERATION

There is no changes in the scheduling phase.

No PL/I code is generated for the statements with DEPENDS_ON function. In routine GENASSR (called from CODEGEN), whose function is to convert a given MODEL equation into a PL/I statement, a conditional RETURN statement is added. It tests the existence of "DEPENDS_ON" and returns back to CODEGEN if "DEPENDS_ON" is found in the text.

CHAPTER 14

THE MAIL AND POST FILES

14.1 FUNCTION

Two new file organizations, MAIL and POST, are added to the previous MODEL language file organizations: SAM and ISAM. The user continues to view the external environment purely in terms of data. However, if the data connects the module to other modules and does not need to be stored as a whole then it is declared as having MAIL organization. Otherwise it is declared as a SAM file. The records are queued (making up a multiple dimension array) in a MAIL file in the order of arrival. The MAIL file thus serves the function similar to a mailbox. The POST file organization is used if a record contains an address of the destination mail file. Thus it is similar to the way post offices distributing messages to mailboxes. A POST file is thus connected to multiple MAIL files. There can be multiple producers and consumers of data of a MAIL file, however, there can be only a single producer of data of a POST file.

14.2 SYNTAX ANALYSIS

The augmented BNF of the FILE declaration in MODEL is as the follows.

```
<FILE_DCL> ::= 1 <NAME> IS FILE [, ORG: <ORG>] [,KEY <NAME>] ,
    <ORG>   ::=   SAM | ISAM | MAIL | POST
```

FIGURE 24. Syntax of a file declaration in MODEL

SAM and ISAM are the two existing file organizations in the old MODEL language. The KEY option is only used in a) a POST file to indicate the field in a record that contains the destinations of the records; or b) an ISAM file to indicate the field according to which a record is accessible.

The syntax analysis program stores in a dictionary the attributes of each symbol. This is done by the semantic routines during syntax analysis. The data structure of the dictionary is named ATTRIBUTES having the following fields.

```
DCL 1 ATTRIBUTES BASED (ATTR_PTR),
      2 XDICT CHAR(32),
      2 XDICTYPE CHAR(4),
      2 XMAINASS PTR,
          :
          .
      2 XRECIO     bit(1), /* recio or stream io */
      2 XINXPOS FIXED BIN; /* INDIRECT VECTOR POSITION */
```

XDICT contains the name of the entry in the dictionary. The field XDICTYPE indicates the type of the entry in the dictionary; it may be one of the following:

i) FILE — a FILE node
ii) FLD — a FIELD node
iii) RECD — a RECORD node
iv) ASTX — an ASSERTION node

If XDICTYPE is FILE, the pointer field XMAINASS points to an auxiliary data structure named FILE.

```
DCL 1 FILE BASED(DP),  /* DATA STRUCTURE FOR FILE DCL */
     2 TYPE CHAR(4),   /* ='FILE' */
     2 STMTS FIXED BIN,
     2 $MEMBERS FIXED BIN,
     2 TABULATED FIXED BIN, /* 0= NOTAB, 1= TAB */
     2 DUMMY FIXED BIN,
     2 KEY_FLAG FIXED BIN,  /* 0= NOKEY,  1= KEYED*/
     2 ISAMB FIXED BIN,     /* 0=SAM,1=ISAM,2=MAIL,3=POST */
     2 MEMBERS(N),
      3 $SUB FIXED BIN,
      3 FIRST_SUB FIXED BIN,
      3 SECOND_SUB FIXED BIN;
```

The field ISAMB in the FILE structure contains the code of file organization. The coding process is done by a semantic routine named SVORG3 during the syntax analysis.

```
SVORG3: ENTRY;
/* SAVE FILE ORGANIZATION INTO ASSOCIATED DATA AREA */
   IF LEXBUFF='ISAM' THEN FILE.ISAMB=1;
   ELSE IF LEXBUFF='MAIL' THEN FILE.ISAMB=2;
   ELSE IF LEXBUFF='POST' THEN FILE.ISAMB=3;
RETURN;
```

Where LEXBUFF contains the symbol recognized by a lexical analyzer. Fully expanded EBNF description of the concurrent MODEL language can be found in Appendix B.

14.3 CODE GENERATION

There is no changes in the array graph analysis, propagation and scheduling phases. Some knowledge of VMS PL/I is required in this section [VAX, 80].

The code generation phase (CODEGEN) consists of searching the entries in the flowchart produced by the scheduler (SCHEDULE), one by one, and interpreting them into PL/I code. Source file entries are transformed into OPEN operations; target file entries into CLOSE operations; source records into READ operations and target records into WRITE operations. The transformation process varies according to two attributes of the node: file organization and target/source. Detailed description of the translation process is presented in the next four sections.

14.3.1 THE OPEN PROCESS

As shown above, the ISAMB field in data structure FILE indicates the file organization of a given symbol. The following table depicts the interpretation of the MODEL compiler to the different file organizations.

Let "name" be the name of the file with type ISAMB in the table.

ISAMB	ORG.	S/T	DESCRIPTION OF PL/I CODE
0	SAM	S	Simple OPEN statement. Option: INPUT, SEQUENTIAL.
0	SAM	T	OPEN process is omitted.
1	ISAM	S	OPEN with read_only options: INPUT, SEQUENTIAL, ENV(SHARED_WRITE).
1	ISAM	T	OPEN with update options: KEYED, SEQUENTIAL, UPDATE, ENV(SHARED_WRITE).
2	MAIL	S	i) Create a mailbox named "name"S_MBX using VMS utility SYS$CREMBX. ii) Obtain the physical name bound to the logical file name: "name" by using the VMS utility SYS$TRNLOG. iii) OPEN the file with obtained physical name.
2	MAIL	T	No need of an OPEN process. The file is open to write by default.
3	POST	T	No need of an OPEN process. The file represents only intermediate distribution.

TABLE 6. Interpretations of a Source File node (OPEN)

REMARKS ON MAIL FILE:

The mailbox creation statements in the OPEN process are designed to "compensate" the mailbox creation process done by the Configurator. In general, a physical mailbox is created for each source mail file by the main JCL program generated by the Configurator. The mailbox is deleted when the module that uses the mailbox as source terminates. However, a module may also be initiated manually and repeatedly. In such cases, the mailbox deleted in the previous run has to be re-created when a module is re-initiated. The mailbox creation statements in the OPEN process can re-create the mailbox if it is deleted in previous run. If the mailbox exists before the execution of the module, the mailbox creation statements in the module become redundant. This may happen, normally, only when a "+" module is initiated. In VMS, however, a redundant mailbox creation request is returned immediately instead of creating a new version of mailbox. The "+" module can still get access to the existing mailbox which may

contain some messages sent by other modules before its execution. Thus the correctness of the implementation is guaranteed.

The logical name translation process is to obtain the physical file name assigned at runtime. If a source MAIL file is assigned to a disk file at runtime, the computation proceeds by consuming the disk file, as if a mailbox file were used. This facilitates local debugging for individual modules by avoiding modification and re-compilation of the individual MODEL programs.

14.3.2 THE CLOSE PROCESS

The following table depicts the CLOSE process for a file node.

Let "name" be the name of the file with type ISAMB in the table.

ISAMB	ORG.	S/T	DESCRIPTION OF PL/I CODE
0	SAM	S	CLOSE process is omitted.
0	SAM	T	Simple CLOSE statement.
1	ISAM	S	CLOSE process is omitted.
1	ISAM	T	Simple CLOSE statement.
2	MAIL	S	Deletion of the mailbox created in the OPEN process using the VMS utility SYS$DELMBX.
2	MAIL	T	No need of a CLOSE process.
3	POST	T	No need of a CLOSE process.

TABLE 7. Interpretations of a Target File Node (CLOSE)

In general, the CLOSE process is performed only for the target files in a module; but in the case of a source MAIL file, a special mailbox deletion process is also necessary.

14.3.3 THE READ PROCESS

The READ process is the interpretation of a source RECORD entry in a schedule. The interpretation varies based on the organizations of the file to which the record belongs.

The following table depicts the interpretations according to the file organizations and their input/output status.

ISAMB	ORG.	S/T	DESCRIPTION OF PL/I CODE
0	SAM	S	Simple READ statement.
1	ISAM	S	Indexed READ statement.
2	MAIL	S	Simple READ statement.*

TABLE 8. Interpretations of a Source Record Node(READ)

*NOTE: The MAIL file has the same READ process as for a SAM file; this is to allow device independence.

Although having the same form as for a sequential file, a READ operation will "wait" if it is issued to read a mailbox and there is not data in the mailbox.

14.3.4 THE WRITE PROCESS

The WRITE process is only for target and update files. Let "name" be the name of the file with types depicted in the following table.

ISAMB	ORG.	S/T	DESCRIPTION OF PL/I CODE
0	SAM	T	Simple WRITE statement.
1	ISAM	T	Keyed WRITE statement.
1	ISAM	S&T	Keyed REWRITE statement.
2	MAIL	T	i) Obtain physical file name from the logical name ii) Assign communication channel to the physical name a) If channel assignment is successful, then WRITE the message to the physical mailbox using VMS utility SYS$QIO b) If no channel can be assigned, WRITE the message into a disk file named "name".DAT iii) Deassign the channel.
3	POST	T	i) Assign channel to the file name contained in the address field of the POST file; a) If a channel is assigned successfully, then WRITE message using the VMS utility SYS$QIO; b) Otherwise WRITE message into a disk file named "name".DAT. ii) Deassign the channel.

TABLE 9. Interpretations of a Target Record Node(WRITE)

REMARKS ON MAIL FILE:

The runtime physical file name obtained at step (i) can be either a mailbox name or a disk file name depending on the logical name assignment in JCL before execution. If a disk file name is found, the WRITE process simply writes the record to the disk file. Otherwise the VMS utility routine SYSSQIO is used to send record to the designated mailbox. For each WRITE action, the channel assignment and deassignment are respectively executed once; this is to allow efficient use of communication channels available on a particular host computer.

REMARKS IN POST FILE:

Instead of using the logical file name: "name" to acquire communication channel, as done for a MAIL file, the POST-file WRITE process acquires channel to the address field in the POST file. This realizes the effects of runtime distribution of records to different destinations.

14.3.5 ON ENDFILE

Using the concurrent programming facilities of VMS-PL/I, the exception handling must be considered. The exceptions are signaled in different ON-units and one can use the exception codes for appropriate action.

A module consuming MAIL file(s) is designed to disregard all the ENDFILE marks it may receive from the mail producers. Because every producer executes a CLOSE statement when it finishes local computation. The effect of executing a CLOSE statement is to put an ENDFILE mark into the mailbox, such that the receiving module can be informed of the termination of a producer module. These marks are not used in the current implementation.

An ON ENDFILE unit is used to recover from reading an ENDFILE mark. Before each READ statement for a MAIL file, there are a label statement and a label assignment statement, shown as follows.

```
$RD_L"recname";
  $RD_L=$RD_L"recname";
  READ FILE("name")...
```

In the ON ENDFILE("name") clause, we have the following:

```
ON ENDFILE("name") BEGIN;
   GOTO $RD_L;
END;
```

Where "name" is the name of the MAIL file and "recname" is the record name in the file. When an ENDFILE condition is signaled, the control of the program is transferred to the ON-ENDFILE unit. The GOTO statement forces the control back to the original READ statement which triggered the ON unit. Since one source MAIL file may contain more than one RECORD structure which will be interpreted as more than one READ statement, there may be more than one READ statement that may signal an ENDFILE condition. The use of the label variable $RD_L and the label assignments ensures the proper trace of the READ statements.

CHAPTER 15

CONCURRENT UPDATE OF ISAM FILES

15.1 PROBLEMS AND OBJECTIVES

Sharing ISAM file concurrently requires the use of record locking mechanism. The VMS-PL/I compiler offers the following automatic record locking facilities.

A record is locked when both of the following are true:

* A READ statement is issued for the record.
* The file containing the record was opened with the OUTPUT or UPDATE attribute.

A record remains locked until one of the following occurs:

* The locked record is rewritten or deleted.
* A READ, WRITE, REWRITE, or DELETE statement is executed to access another record in the same file.
* The REWIND built-in subroutine is called to rewind the file to its beginning.
* The file is closed.

Records are also locked for the duration of a WRITE, REWRITE, or DELETE statement to ensure that the I/O completes. The records are unlocked when these statements complete.

If a module attempts to access a locked record, an ERROR condition will be signaled. The condition can be sensed in an ON ERROR unit (similar to an ON ENDFILE unit).

In majority cases, an ISAM file is accessed one record at a time. The above facilities are sufficient to provide protection to the shared data in such cases. However, in more complex applications, a

module may want to lock several records in order to update them correctly, meanwhile allowing other modules to access other records in the same file. This kind of application will be called here "multi-record access". Since the available facilities at hand (VAX/11-PL/I and VMS) do not have the multi-record locking ability, the MODEL compiler detects whether a multi-record access is implied by the specification, it issues a warning to the user confirming that mutual exclusion will not be provided automatically.

The MODEL compiler is modified to attempt to schedule the READ and WRITE or REWRITE processes of an update ISAM file inside one loop. This ensures that if possible, a user's specification will be scheduled into one-record access.

15.2 SCHEDULING

There is no changes in the stages before scheduling.

In scheduling stage, a modification is made to force the merge of a READ and a WRITE operation of an update ISAM file into one loop for mutual exclusion in updating shared records.

To illustrate the idea, let us consider the following example. If X is an update ISAM file, a MODEL specification which manipulates the file may produce the following different flowcharts:

```
a) Do loop 1;
      :
     READ FILE(X);
      :
   End;
   :
   Do loop 2;
      :
     REWRITE(or WRITE) FILE(X);
      :
   End;
```

```
b) Do loop 1;
      :
     READ FILE(X);
      :
     REWRITE(or WRITE) FILE(X);
      :
   End;
```

If the module is running concurrently with other modules, schedule (b) is more desirable due to its implicit use of VMS-PL/I one record locking facilities (see section 15.1).

In principle, the possibility of merging two loops together is decided by the use of subscripts in a MODEL specification and an optimization algorithm. The optimization algorithm first determines all the possible mergers of components in the array graph and then calls a function EVALUATE to compute the memory penalty of each merger. The merger with smallest penalty will be chosen and produced in the flowchart. To force the merger of a READ and a REWRITE or WRITE operations (RW-merger), the function EVALUATE is modified. The modified EVALUATE first checks the existence of a READ and a REWRITE(or WRITE) of an update ISAM file. If found, the function returns (- 10,000,000 + real penalty). Here 10,000,000 is assumed to be the maximum memory penalty for a merger. This will force a RW-merger with smallest memory penalty to be chosen, i.e. if more than one RW-merger have been found, the one with smallest penalty will be chosen.

After scheduling, a flowchart of the MODEL specification is constructed. In procedure "PREPARE" (in Figure 23), the check of multi-record accesses on ISAM files is performed. It is done by scanning the flowchart looking for consecutive READ (X) statements (without REWRITE or WRITE in between), where X is a update ISAM file. A warning message will be issued, if such READs have been found.

15.3 CODE GENERATION

When a module attempts to access a blocked record (i.e. being updated by another module) then an error condition is created. The objective of the following generated code is to recover from the error and attempt another access later.

i) Before each READ, WRITE, REWRITE, and DELETE statement, there are a label statement and a label assignment statement. The labels before different statements are named differently. This is to keep track of a statement which signals the condition. The following figure shows a sketch of the produced code.

```
LS"name":
$RD_LP$=LS"name";
READ FILE("name") INTO(...) KEY(...);
```

```
$RW_"name":
$RD_LP$=$RW_"name";
REWRITE FILE("name") FROM(...) KEY(...);
```

```
WS_"name":
$RD_LP$=WS_"name";
WRITE FILE("name") FROM(...) KEYFROM(...);
```

```
DS_"name":
$RD_LP$=DS_"name";
DELETE FILE("name") KEY(...);
```

FIGURE 25. PL/I code for concurrent ISAM file accessing

ii) In the generated ON ERROR unit, a goto statement is added (in procedure CODEGEN):

```
ON ERROR BEGIN;
   IF ONCODE()=RMS$_RLK THEN GOTO $RD_LP$;
                :
                .
END;
```

FIGURE 26. The ON-UNIT for concurrent ISAM file accessing

The RMS$_RLK is the condition signaled by the access to the locked record; it stands for "Record Management System_Record LocK".

CHAPTER 16

ITERATIVE SOLUTION FOR DISTRIBUTED SIMULTANEOUS EQUATIONS

16.1 PROBLEMS AND OBJECTIVES

DSE (Distributed Simultaneous Equations) is a system of simultaneous equations which span more than one module. For the system is to be solved iteratively, it should terminate an iteration only if all of its modules are converged (or have run out of iteration limit). Otherwise incomprehensible results may be produced.

The objective of the modification to the MODEL compiler is to design an algorithm that can be attached to the produced PL/I program and can simultaneously control the termination of each iteration of every module.

The problem is equivalent to the well know "distributed termination problem" studied in [Dijkstra, 83] and [Francez, 82]. Formally, it can be stated as follows. Given an arbitrary distributed system consisting of N modules that are strongly connected and run in parallel, the system termination condition is:

$$B = B1 \ \& \ B2 \ \& \ B3 \ \& \ \ldots \ \& \ BN.$$

‹Bi› is called the "stable" or "termination" condition for module Mi. The objective is to terminate the system as soon as all ‹Bi›s are satisfied. For a DES system, since all the modules are computing synchronically, the stable condition of each module may be indexed by the iteration number t. Consequently, the termination control problem can be stated as to find the smallest t, such that

$$B(t) = B1(t) \ \& \ B2(t) \ \& \ B3(t) \ \& \ \ldots \ \& \ BN(t)$$

becomes true and to terminate an iteration of all the modules at exactly the same t.

16.2 SOLUTION TO DISTRIBUTED TERMINATION PROBLEM

To control the termination of a distributed system, a termination control algorithm must be bound to each individual module; but in no case should the original computation of each module be altered.

16.2.1 COMPARISON WITH THE KNOWN SOLUTIONS

There are two important differences between our solution and the ones studied in [Dijkstra, 83] and [Francez, 82].

i) Network connection.

In [Francez, 82], an undirected spanning tree is chosen in connecting a distributed system. In [Dijkstra, 83], an undirected star-like network structure is used. In our solution, however, the network connection can be a SCDG (Strongly Connected Directed Graph). Both spanning trees and the star-like networks are special cases of SCDG if the undirected edges are treated as pairs of directed edges.

ii) Symmetric treatment of modules

The requirement of knowing the "root" of the connection tree or the "center" of a star structure is dropped to allow every module to be equally treated and the solution to be symmetric. Consequently, the same control algorithm is applicable to every node (module) in the SCDG.

16.2.2 ASSUMPTIONS OF THE TERMINATION CONTROL ALGORITHM

Following are the assumptions of the termination control algorithm.

i) All the modules participate in a SCDG.

ii) Every module Mi will eventually satisfy its corresponding ‹Bi›, and once the ‹Bi› of a given module and all its predecessors' are satisfied, that module will keep its ‹Bi› satisfied.

iii) There is only one format for control messages; i.e., the control format of a module's output must be its successors' input control message format.

iv) The maximum diameter (D) of the SCDG is known in advance.

Assumption (iv) may not be necessary if greater communication overhead can be tolerated to some extent (see section 16.2.4).

16.2.3 THE TERMINATION ALGORITHM AND ITS DERIVATION

In the sequel we use standard terminology of graph theory [Aho, 74]. By the distance L(x,y) from node x to y, we mean the length of the shortest (directed) path between them. The network diameter (i.e. the greatest distance between any two nodes) is denoted by D.

The termination detection algorithm involves sending tokens through the network. Tokens have integer values from the interval ‹0,D+1›. They are transmitted as a part of communication traffic generated by the main computation. Thus, in the case of distributed solution of simultaneous equations each message sent between processes consists of a solution of local equations and a token representing the state of a node.

Each process can count the number of times it has received messages from all of its predecessors. We will use this value as a local index of node's local activities, i.e. reading input, performing computation and sending output. It is worthwhile to note that in the considered case the main computation is partially synchronized, because no process can complete receiving its t+1-st input messages before all the processes have received their t-th input messages. It also means that in order to synchronize deactivation of processes, each one of them has to stop with the same value of the

local index.

Let T(t,x) denote a predicate indicating whether the local termination conditions are satisfied in node x for local index i. We assume that initially T is false, i.e. for any node x, T(0,x)=F.

By I(t,x) we will denote the minimum value of tokens received by the node x in the input indexed by t. Let S(t,x) denotes the t-th state of the node x defined as:

$$S(t,x)=\begin{cases} 0 & \text{- if } T(t,x)=F \\ \min(S(t-1,x)+1, I(t,x)) & \text{- otherwise} \end{cases}$$

The state is well defined because for t=0 T(0,x)=F and therefore S(0,x)=0. The output token of node x sent out with index t is equal to S(t,x)+1. (This token is received by successors of node x as a part of their t+1-st input).

Informally speaking a node x is a generator of new "0" tokens when T(t,x)=F (i.e. it is not ready to stop) and it is a transmitter of tokens otherwise. In the latter case the node selects the minimum token among its actual input and previous output, and then increases it by one and sends it further to the network. When all the nodes are ready to stop, all are transmitters and no new "0" tokens are generated. Therefore the transmitted tokens grow in value, as do the node states. We show below that the network diameter is a limit value which a node state can exceed only after all the nodes are ready to stop. All nodes reach that state with the same local index.

To show this more formally, we notice first that the definition of the node's state implies that

$S(t,x) \leq S(t-1,x)+1 \leq t$ for any node x and index $t>0$.

As usual, the system state will be captured by an invariant, for instance Q, defined as:

Q: For any index t and node x if $S(t,x)>0$ then

$(\forall y:L(y,x)<S(t,x),\forall j:t-S(t,x)<j\leq t-L(y,x)) :: S(j,y)>0$

or in other words, for any node y and any index j if $t-S(t,x)<j \leq t-L(y,x)$ then the node y is ready to stop for the index j, i.e. $T(j,y)=T$. The condition $S(j,y)>0$ to be well-defined requires $j\geq 0$, but this follows from inequality $t\geq S(t,x)$ holding for any node x and index $t>0$.

Now, we show that the traffic of the tokens described above keeps Q true. Q is true for index $t=0$, because for any node x, $S(0,x)=0$, by definition. Suppose that Q does not hold for a node x. Let $i0>0$ denote the smallest index of such event, and y0 denotes a node violating Q. Thus, we have

$(\exists j:i0-S(t0,x)<j\leq t0-L(y0,x)) :: S(j,y0)=0.$

Let j0 denote such j. Since Q holds for t0-1, we have

$(\forall j:t0-S(t0-1,x)-1<j\leq t0-L(y0,x)-1) :: S(j,y0))>0.$

Since $S(t,x) \leq S(t-1,x)+1$ for any index $t>0$ and node x, then j0 may be only equal to $t0-L(y0,x)$, and traversing the shortest path from y0 to x we can start with the index $t0-L(y0,x)$ and the state $S(t0-L(y0,x),y0)=0$, to reach x with the input indexed by t0 and such that $I(t0,x)\leq L(y0,x)$. But the definition of the node's state implies that $S(t0,x)\leq I(t0,x)$, what contradicts the assumption that $S(t0,x)>L(y0,x)$. The contradiction proves that Q is indeed an invariant.

If for any node x and index t, $S(t,x)>D$ then from Q we conclude that for any node y the inequality $S(t-D,y)>0$ holds (since $L(y,x)\leq D$) and the entire computation is in a stable state.

Now we have to show only that if for some index t, and node x, $S(t,x)>D$ then for any node y also $S(t,y)>D$, i.e. that the stable state of computation is recognized in all the nodes synchronically.

This is implied by another invariant R, defined as:

R: For any node x and any index t there is such a node y that $S(t-S(t,x),y)=0$.

If $S(t,x)=0$ then R holds by setting $y=x$. When $S(t,x)>0$ we can always construct a sequence of nodes $y(0)=x$, $y(1)..y(k)$, $k=S(t,x)$ such that for $l=1,2...k$.

$$S(t-l,y(l))=S(t-l+1,y(l-1))-1$$

To do that, let's consider evaluation of $S(t-l+1,y(l-1))$. If $S(t-l+1,y(l-1))=S(t-l,y(l-1)+1$ then we set $y(l)=y(l-1)$. Otherwise as the element $y(l)$ we select the node which sends the token with the minimum value for the $t-l+1$-st input of the node $y(l-1)$. The last node in the constructed sequence, i.e. $y(S(t,x))$, satisfies the condition of the invariant R.

If some nodes x,y and index t satisfies the conditions $S(t,x)>D$ and $S(t,y)\leq D$, then from R we conclude that there exists such a node z that $S(t-S(t,y),z)=0$. But from Q, based on $S(t,x)>D$, we have that $S(t-k,v)>0$ for any node v, and any $k\leq L(v,x)\leq D$. The contradiction proves that all the nodes of the network reach the state D+1 with the same local index. Therefore reaching this state can serve as a trigger for deactivation of the corresponding process. It is easy to verify that in fact we can use any value greater than D as a trigger for deactivation, paying price however of continuing the main computation unnecessarily. D+1 is in fact the smallest trigger value independent of the pattern of getting nodes ready to stop.

Finally, we can give the termination algorithm as follows.

ALGORITHM 12. DISTRIBUTED TERMINATION CONTROL.

Let D be the diameter ,
(Ti) be the set of input tokens of Mi and Oi is the output token of Mi, and
PRE_Oi be the value of the previous (t-1) output token Oi.

Initialization: PRE_Oi=1.

```
0.   If ‹Bi›
1.   then if PRE_Oi ‹ MIN({Ti})
2.        then Oi = PRE_Oi + 1.
3.        else Oi = MIN({Ti}) + 1.
4.   else Oi=0.
5.   If PRE_Oi › D +1 then stop.
6.   PRE_Oi = Oi.  □
```

The implementation of the algorithm is described in section 16.3.

16.2.4 FINDING THE DIAMETER OF THE NETWORK DYNAMICALLY

Lack of knowledge of full network topology at compilation time implies that the network diameter may be unknown until run-time. Although the current implementation uses the Configurator to calculate the diameter, this section provides a simple algorithm which may reside in each module and compute the diameter at runtime without using a centralized configuration.

For the given network we denote the number of its node by n and its connectivity matrix by M. Let M1 denote a (n x n) matrix with all the entries equal to 1. The network diameter is the smallest exponent D such that M to power D yields M1. Each node in the network initially knows only its predecessors, i.e. one row of the matrix M. To minimize the communication traffic we want to propagate the smallest possible amount of data. Thus, instead of attempting to build up entire matrices of powers 1,2...D of M, in each node we will construct only a single vector representing in a step k a row of the k-th power of M. We denote the elements of the connectivity matrix M by m(x,y) x=1,2...,n, y=1,2...,n, where m(x,y)=1 means as usual that the node x is a predecessor of the node y. The vector m(k)(-,y)

representing the k-th power of M in the y-th node is equal to

$$m(k)(-,y)=\sum_{z=1}^{n} m(k-1)(-,z)*m(z,y)=\sum_{m(z,y)=1} m(k-1)(-,z)$$

But m(z,y)=1 means that the node z is the predecessor of the node y. Hence constructing such a vector requires only sending to the given node vectors of the previous power of M from this node's predecessors. The existing communication links can be readily used to carry that task.

To further decrease the communication traffic we can send only this part of a vector m(k-1)(-,z) which contains 1's not sent in the previous steps and not to store 0's at all. In fact, initially each node knows only 1's of its own vector. Thus, in each step k we have to send names of nodes by which the vector m(k-1)(-,z) was extended in the previous step.

A node y can recognize that it has constructed the entire vector m(k)(-,y) when in some step k all the node names received have already reached the node y before. Indeed, if there exists a node x such that L(x,y)>k-1 then on the shortest path form x to y there is a node z such that L(z,y)=k. Therefore the name of z reaches the node y at the k-th step (and never before) contradicting the assumption about the step k. The number of unique names received until the step k is equal to the network size n, and the longest distance in the network from any node to the given node y is equal to k-1. We will refer to that latter value as a relative diameter D(y) of the node y.

In quite a similar way to the termination state token propagation the nodes can propagate also tokens representing the biggest distances found. Each node sends out the maximum value of distances received on input and its own relative diameter (or step number, if the relative diameter is not evaluated yet).

The network diameter D is equal to such node x's output token that is first repeated after the step 2*D(y).

Following is the complete algorithm for finding the network diameter D.

ALGORITHM 13. DISTRIBUTED DIAMETER EVALUATION FOR A NODE X.

INPUT FORMAT: [k,N1,N2,...,Nn,T](i)

where N1(i),...,Nn(i) are the input module identifiers received at the i-th input by module X, k(i) is the number of identifiers (module names) received and T(i) is the biggest distance token.

For every node x, attach the following:

Let D be the diameter of a global network,
CNT be the index counter (t),
Dx be the local diameter value of node x,
{ID} be the set of all the input module names received by node x,
{T} be the set of all the distance tokens received by node x,
TO,PRE_TO be the output distance token and the previous output distance token respectively,
STK be a stack of identifiers collected from the input,
NEW_ID(y) be a Boolean function which returns "true" if y is a new identifier to node x, and
{OID} be the set of output identifiers.

INITIALIZATION: CNT=1. D=0. Dx=0. PUSH(x) onto STK. {OID}={x}.

```
1. New='0'B.
2. Do the following for every input i of node x:
3.    Do the following for j = 1 to k(i):
4.       If NEW_ID(Nj(i)) then PUSH(Nj(i)), New='1'B.
5.    End.
6. End.
7. If New then Dx=CNT-2.
8. If Dx=0 then TO=CNT, else TO=MAX(Dx,{T}).
9. If CNT>Dx*2 & PRE_TO=TO then D=TO.
10.PRE_TO=TO.
11.If D=0 then set {OID} = STK, n = the length of STK.
          else set {OID} = Ø, n = 0, STK = empty.  □
```

To verify the correctness of that rule, let v denote such a node that D(v)=D. Since L(v,y)<=D(y) then in the step D(y)+L(v,y)+1 the token D(y)+1 sent from v reaches y. From that step on the output tokens from y will grow until they reach D.

According to that rule, every node v will know that D(v)=D at the step 2*D+1.

•

As we use the existing communication links to accomodate traffic of signals created by this algorithm we can attached these signals to the messages communicated by the main computation. Therefore the steps of the algorithm for finding the network diameter will

correspond to local indexes of the termination detection algorithm. Consequently the first instant at which the entire computation for finding diameter can be stopped corresponds to the index 2*D+1.

The entire termination detection algorithm, including finding the network diameter, consists of three phases. At the beginning, for indexes 1 to D three streams of signals are flowing through the network:

1. Termination state tokens.

2. Names of nodes for constructing connectivity vectors. This stream gradually dies as the nodes complete building their vectors.

3. The biggest distance tokens.

Then for indexes D+1 to 2*D+1 two streams, the first and the third, are active. Finally, for indexes greater than 2*D only termination state tokens flow as all the nodes know the diameter D and the auxiliary algorithm of finding the diameter stops.

The distributed diameter finding algorithm involves additional communication overhead (stream 2 above). Therefore the use of the diameter computed during configuration compilation by the Configurator is preferred. But it is justified only when every module in the DSE system participates in only one DSE. Otherwise the configuration may represent a merger of all the individual DSE's involved. Such a merger may have smaller diameter than some of its constituents.

As the distributed diameter finding algorithm is not generated by the current implementation, it is the user's responsibility to provide correct diameter (the maximum value of all the possible diameters) through a CSL specification, if necessary.

16.3 GENERAL DESCRIPTION

To give an idea of how to specify a module involved in a DSE system, we first present an example of a set of nested simultaneous equations without involvement with a DSE system:

```
 1.  MODULE: NESTMOD;
 2.  SOURCE: NESTIN;
 3.  TARGET: NESTOUT;

 4.  1 NESTIN FILE,
       2 NESTREC RECORD,
         3 (K1,K2,K3,K4) FLD (PIC 'B9.V00');

 5.  1 NESTOUT FILE,
       2 OUTREC RECORD,
         3 (X,Y,A,B,C,D) ARE FLD (PIC 'BBS(5)9.V(5)9');

 6.  BLOCK BLK1: MAX ITER IS 100;
 7.    X = A * Y + B;
 8.    BLOCK BLK2: MAX ITER IS 100;
 9.      A = 0.2 * B + K1 + X -X;
10.      B = 0.2 * B + K2;
11.    END BLK2;                        (to be continued)

12.    Y = C * X + D;
13.    BLOCK BLK3: MAX ITER IS 100;
14.      C = 0.2 * D + K3 + Y - Y;
15.      D = 0.2 * C + K4;
16.    END BLK3;
17.  END BLK1;
```

By adding the assertion: (K1,K2,K3,K4)=DEPENDS_ON(X,Y,A,B,C,D) after line 6, NESTMOD becomes a concurrent module participating in a DSE system. The DEPENDS_ON statement indicates that there is an external "environment" that computes K1, K2, K3 and K4 based on X, Y, A, B, C and D in some unknown way. Module NESTMOD cannot terminate an iteration if the convergence condition of the external "environment" is not satisfied. In other words, the presence of a DEPENDS_ON statement is a necessary condition of a module's participation in a DSE system.

In an array graph (see section 5.1), a set of simultaneous equations is represented as an "undecomposable" MSCC. A scheduler can unravel the MSCC and produce an iterative procedure to solve it. A MSCC can be unravelled in many different ways each of which leads to a differently blocked structure. Since the equation written sequence

affects the speed of convergence and stability of the system, the MODEL compiler unravels a MSCC according to the equation written order. The order imposed by data dependency is also followed in a unravelled MSCC. This gives the developer of the system some freedom in adjusting the speed of convergence as well as stability of the solution process.

A BLOCK statement in MODEL allows a user to specify the solution method (currently Gauss-Seidel), relative error, maximum iteration limit and nesting structure of the iterative procedures. A user can use the block statements to identify the dense clusters in a simultaneous equation system.

The presence of an external dependency statement in a MSCC is a necessary condition of the MSCC's involvement in a DSE system, therefore it changes the translation process for the module.

16.4 SCHEDULING

There is no changes made in the stages before scheduling.

There are two major recursive procedures in the scheduler: SCHEDULE_GRAPH and SIMUL_BLK. The modifications are made in procedure SIMUL_BLK.

Every BLOCK statement has a corresponding block structure created during syntax analysis. Each block structure contains all the information specified in the BLOCK statement and the scope of the block. The MODEL compiler also creates, automatically, a "universal block" which scopes from zero to the maximum statement number reached during the syntax analysis of a MODEL specification.

Procedure SCHEDULE_GRAPH can topologically sort an array graph and produces flowchart for the sorted elements. It calls procedure SIMUL_BLK, if a multi-node MSCC is found in the graph. SIMUL_BLK unravels the nested block structure in a MSCC. Following is the description of procedure SIMUL_BLK.

Algorithm 14. MSCC UNRAVELLING (Procedure Name: SIMUL_BLK)

Input : A MSCC, A list of BLOCK statements
Output: A flowchart of the unravelled MSCC with proper block structure

```
 1. Find the first and last statement numbers in the MSCC.
 2. Find the smallest BLOCK cur_blk which contains the
    entire MSCC.
 3. Set dpds=nil.
 4. For every element e in the MSCC do the following:
 5.     If e is a DEPENDS_ON statement
 6.     then do;
 7.       Set nesting=0.
 8.       For each member b in the BLOCK list do the following:
 9.         If the b's begin statement  ≥ cur_blk's begin statement
               and b's end statement  ≤ cur_blk's end statement
10.         then do;
11.           if the statement number of e > b's begin statement
12.           then nesting=nesting+1.
13.           If the statement number of e > b's end statement
14.           then nesting=nesting-1.
15.         End.
16.       End.
17.       If nesting < 2 then do;
18.         set dpds=e.
19.         Cut the edge from e (a DEPENDS_ON statement)
                to its target variable(s).
20.       End.
21.     End.
22. End.
23. If dpds=nil then do;
      /* find the First Element FE and cut backwards edges */
24.   If the first statement in the current block is a BLOCK statement
25.   then set FE to be the entire BLOCK.
26.   else set FE to be the first statement in the current block.
27.   Do the following for each element e in the MSCC.
28.      If e does not belong to FE and there is an edge emitting
            from e to FE then cut the edge.
29.      If e belongs to FE then if there is an edge emitting from e
            to e, cut the edge.
30.   End.
31. End.
32. If the cur_blk bears no tag or bears a tag "$EXT"
33. then do;
34.   If dpds≠nil
35.   then mark the tag of cur_blk "$EXT"
36.   else mark the tag of cur_blk "DONE".
37.   Collect all the direct or indirect target FLD nodes
        as the initialization variables for this MSCC.
38.   Create a header of a simultaneous block for this MSCC.
39.   Recursively call SCHEDULE_GRAPH to schedule the modified MSCC.
40.   If dpds=nill then Set the cur_blk tag = "     ".
41.                else set the cur_blk tag = "$EXT".
42.   Create an END mark of the simultaneous block in the flowchart.
43. End.
44. Else recursively call SCHEDULE_GRAPH to schedule the modified
    MSCC. □
```

The above algorithm consists of three major steps:

i) Find proper edge(s) to cut.

This step can be further divided into two parts:

a) Find DEPENDS_ON statements on the current level of BLOCK group nesting.

This is done by lines 1-21. Line 1 determines the scope of the given MSCC. Line 2 finds the smallest block which contains the entire MSCC and establishes the current level of nesting in BLOCK groups (cur_blk). Since the MODEL compiler creates, automatically, a "universal block" enclosing all the statements in a MODEL specification, cur_blk always exists.

Only the DEPENDS_ON statements inside this BLOCK group but not enclosed in nested BLOCK groups are of interest here. Such DEPENDS_ON statements are identified at line 18. Lines 7-17 check the nesting of DEPENDS_ON statement in BLOCK groups. The cutting of the edge from a found DEPENDS_ON statement to its target variable is done in line 19. Its effect is shown in the following figure.

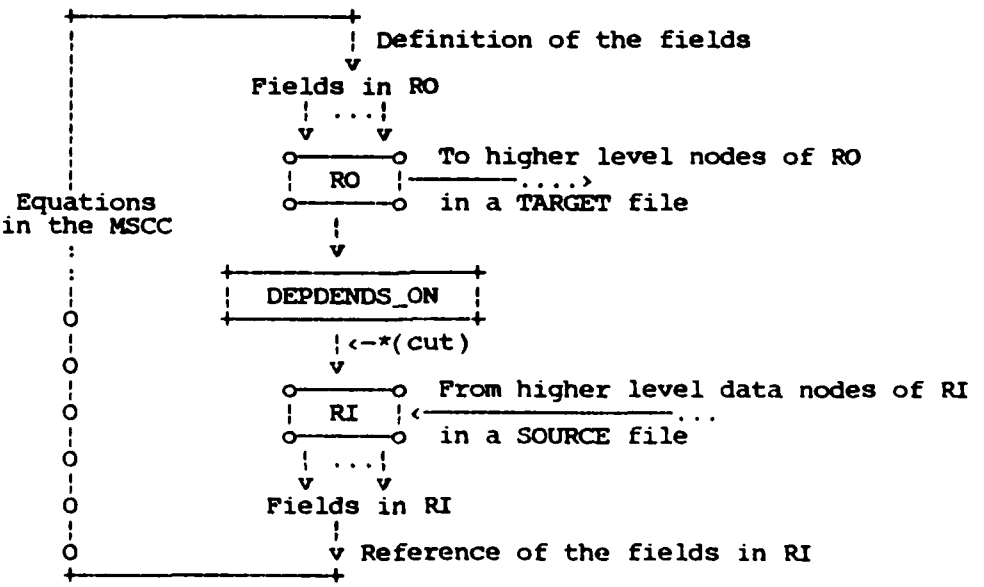


FIGURE 27. Cutting an edge for a MSCC enclosing a DEPENDS_ON statement

Recall that the LHS variable of a DEPENDS_ON statement

must be a RECD (or a higher level) node in a source file and the LHS variables must be RECD (or higher level) nodes in target files. The cutting allows the topological sorter, SCHEDULE_GRAPH, to arrange proper computation sequence according to the dependency in the array graph. The result flowchart of the above array graph after cutting and sorting is as follows.

```
READ RI
unpacking fields in RI
computation
   :
packing fields in RO
WRITE RO
```

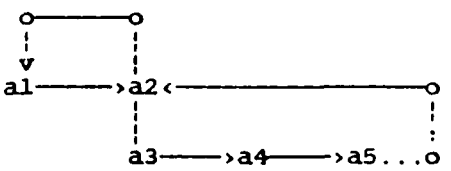
b) Find the first element FE in the given MSCC

This task is performed by lines 23-31 only when no current level DEPENDS_ON can be found. If the first statement in the current block is a BLOCK statement, we have the following MODEL specification:

```
BLOCK BLK1;
  BLOCK BLK2;
    a1
    a2
  END BLK2;
  a3
  a4
  a5
   :
END BLK1;
```

Assertion a1,a2 are identified as FE. Where ai,i=1,2,3,4..., are assertions numbered according to their written sequence.

Taking the first immediate block as FE prevents cutting the edges inside the first block, which will result in a different nesting structure than the one specified by the user. For example, if we have the above MODEL specification with the following array graph:



Cutting the edge from a1 to a2 (or a2 to a1) will leave us a MSCC containing a2,a3,a4,a5... . By using SIMUL_BLK, the following flowchar will result:

```
ITER BLK2;
   a1
   ITER BLK1;
     a2
     a3
     a4
     a5
      :
   END BLK1;
END BLK2;
```

This is not consistent with the structure specified by the user.

If there is no such an immediate block inside of a MSCC, the first element (in the order of written sequence) must be taken as FE.

Lines 27-30 cut the backward edges from the other elements in the MSCC to FE. They also cut e-e type edges for the elements in FE. This allows to unravel unnormalized form of simultaneous equations.

Further unravelling of the modified MSCC is done by a recursive call to SCHEDULE_GRAPH.

ii) Collect a list of variable names needed for initialization.

Each block of iterative procedure needs to initialize all the variables that are LHS' of equations inside the block. Line 37 performs this task. Note that the field variables indirectly involved in LHS of a DEPENDS_ON statement must also be included

in the initialization process.

iii) Creating a block structure in the flowchart, if necessary.

A simul-block structure (type 3) in a flowchart is created only if the entire MSCC is enclosed in a block statement (cur_blk) which bears no tag , or "$EXT" tag. The latter means that there is a DEPENDS_ON statement at the current level.

Note that the relative error, iteration limit and other block information contained in a outer block statement are propagated inward to the automatically generated nested block(s).

Such an unravelling process can effectively find the nested structure of a MSCC involved in a DSE system and automatically attach a block (an iterative procedure) to solve those closely related equations in order to reduce communication cost.

As an example, let us consider the following MSCC consisting of six assertions. We assume that the user did not specify any BLOCK structures.

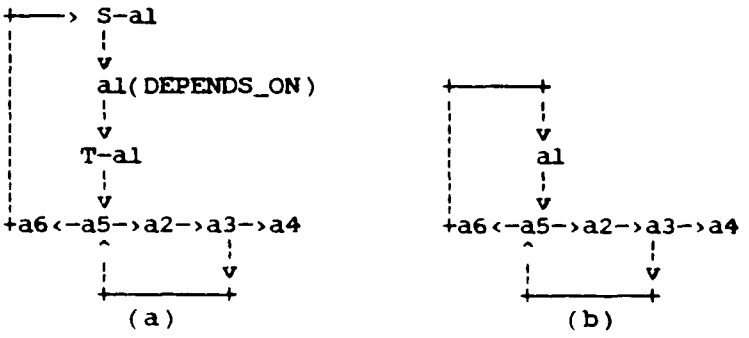


FIGURE 28. MSCCs in an array graph

where ai,i=1...6, are assertions named in their written sequence. Assertion a1 is a DEPENDS_ON statement in (a) but is an ordinary one in (b). S-a1 and T-a1 are source and target (record) variables of a1 respectively.

For the array graph in Figure 28(a), SIMUL_BLK first cuts the

edge from a1 to its target T-a1 by line 22. Then a block is created (the "universal block") and a new MSCC consisting of only a2,a3,a4 and a5 is handed to be scheduled recursively. Procedure SIMUL_BLK called again from SCHEDULE_GRAPH in scheduling the new MSCC, cuts the edge from a5 to a2 (FE) and attaches a block enclosing a2-a5.

The following is the produced flowchart:

```
Block 1
  T-a1
  Block 2
    a2
    a3
    a4
    a5
  End block 2
  a6
  S-a1
  a1
End block 1
```

For the MSCC in Figure 28(b), although it has the same structure as the one in Figure 28(a), the following flowchart is produced (c.f. lines 32-41).

```
Block 1
  a1
  a2
  a3
  a4
  a5
  a6
End block 1
```

Using BLOCK statements, the user can easily create the flowchart similar to the one created for Figure 28(b).

16.5 THE PROCEDURE "PREPARE"

After the scheduling, a new procedure PREPARE is added to attach the control token data fields to be used in the termination algorithm.

The procedure PREPARE has two tasks:

i) collect the target and source record names and corresponding

iteration block numbers in a MSCC of a DSE system

ii) modify the collected records' lengths

The collected record names are stored in a data structure SIMU_NODES in the form of a list. To present the algorithm for scanning the flowchart, it is necessary to present the data structure of the flowchart first.

There are four kinds of data nodes in a flowchart.

a) Simple data or assertion nodes

b) FOR nodes (enclosing a FOR loop)

c) Sublinear block nodes (enclosing a sublinear block)

d) Simultaneous block nodes (enclosing a simultaneous block)

The data structure description for these four kinds of nodes is as follows.

```
DCL 1 NELMNT BASED (NLMNPTR),
      2 NXTNLMN PTR,          /* POINTER TO NEXT FLOWCHART ELEMENT   */
      2 NLMNTYPE FIXED BIN,  /* =1 SIMPLE NODE-ELEMENT               */
      2 NODE$ FIXED BIN,      /* INDEX IN THE DICTIONARY             */
      2 LVL FIXED BIN;        /* LEVEL OF FOR LOOP                   */

DCL 1 FELMNT BASED (FLMNPTR),
      2 NXTFLMN PTR,          /* POINTER TO NEXT FLOWCHART ELEMENT   */
      2 FLMNTYPE FIXED BIN,  /* =2 FOR-ELEMENT                       */
      2 ELMNTLIST PTR,        /* POINTS TO A LIST OF LOOP ELEMENTS   */
      2 FORNAME FIXED BIN,   /* NAME OF THE LOOP CONTROL VARIABLE    */
      2 FORRANGE FIXED BIN, /* LOOP RANGE                            */
      2 VIRINREC bit(1);    /* VIRTUAL OR PHYSICAL                  */
 DCL 1 SELMNT BASED (SLMNPTR),
      2 NXTSLMN PTR,           /* POINTER TO NEXT FLOWCHART ELEMENT */
      2 SLMNTYPE FIXED BIN,    /* =3 SIMUL BLK, =4 SUBLINEAR BLK    */
      2 SLMNLIST PTR,          /* POINTS TO LIST OF BLK ELEMENTS    */
      2 SLMNLABEL CHAR(MAXLENNAME), /* BLOCK NAME FROM USER        */
      2 SLMNLEVEL FIXED BIN,          /* BLOCK NESTING LEVEL       */
      2 SLMNMETHOD FIXED BIN,         /* SOLUTION METHOD CODE      */
      2 SLMNMAXITER FLOAT DEC,  /* MAXIMUM ALLOWABLE ITERATION     */
      2 SLMNRELERROR FLOAT DEC, /* RELATIVE ERROR                  */
      2 SLMNNAME FIXED BIN,      /* ITERATION CONTROL VAR NAME     */
      2 SLMNVARS PTR;            /* INITIALIZATION VARIABLE LIST   */
```

FIGURE 29. Data structures of a flowchart

A typical flowchart skeleton is as follows.

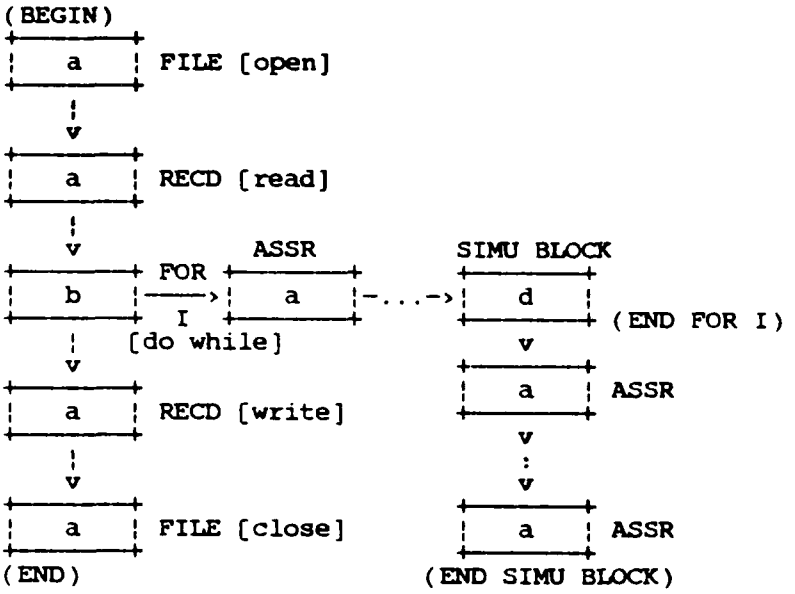


FIGURE 30. Representation of a flowchart

Task (i) is accomplished by the following algorithm.

ALGORITHM 15. FINDING EXTERNAL RECORD NAMES IN A MSCC
(Procedure Name: GENERATE)

```
Input : A flowchart generated by the scheduler
Output: A list of record nodes with corresponding iteration
        level numbers

Let "flowchart" be the pointer to the flowchart produced by
the scheduler (SCHEDULE).
The initial step is to call GENERATE("flowchart",0).

1. GENERATE(root,in_simu_blk) recursive.
2. Do the following for each node in the flowchart from the root:
3.     If the node is a single node
4.     then if in_simu_blk=1  &
            the current node is a DEPENDS_ON assertion
6.          then do;
7.            Put in SIMU_NODE list all the successors and
8.            predecessors of "node", whose attribute is "RECD"
9.            and save the current iteration block number.
10.         end.
11.    else if the node is a simultaneous block node
13.         then call GENERATE(next pointer,1).
14.         else call GENERATE(next pointer,in_simu_block).
15.End.  □
```

The data structure ot SIMU_NODES is as follows.

```
DCL 1 SIMU_NODES BASED(S_PTR),
      2 NDS FIXED BIN,     /* INDEX OF A RECD NODE IN SYMBOL TABLE */
      2 SL_NAME FIXED BIN,/* ITERATION BLOCK (LEVEL) NUMBER        */
      2 NEXT PTR;          /* POINTS TO NEXT SIMU__NODE             */
```

Task (ii) is accomplished by adding 10 to the computed record lengths of the records stored in SIMU_NODES. The physical attachment of the token in the PL/I program is performed in code generation (CODEGEN). To explain the attachment, let XG1R be the record involved in a DEPENDS_ON statement, either LHS or RHS. Then the follwing data structure will appear in the generated PL/I program:

```
1 X FILE,
  2 XR1(*) RECORD,
    3 XF11 ...,
  2 XR2(*) RECORD,
    3 XF21 ...,
  2 XG1(*) GRP,
    3 XG1R(*) RECORD,
      4 XF32 ...,
        :
      4 $IT$ FLD (PIC '9999999999'),
    3 XG2R(7) RECORD,
      4 XF41 ...;
```

FIGURE 31. Representation of a file structure and the patched token field

16.6 CODE GENERATION

16.6.1 ACQUIRING THE DIAMETER OF A NETWORK

As mentioned earlier, the Configurator computes the diameter of a DSE system and places a logical name assignment command in the individual JCL program for each module in a DSE system:

$DEFINE MAX_D "integer".

Where MAX_D is a logical name and "integer" is the computed diameter. The value of MAX_D is then transferred into individual modules at runtime.

In the beginning of each PL/I program generated for the modules involved in the DSE system, there are statements for acquiring the diameter:

```
LN_='MAX_D';
EQV_NAME='';
STS$VALUE=SYS$TRNLOG(LN_I1USAS,L_LENGTH,EQV_NAME,,,);
MAX_D=EQV_NAME;
```

FIGURE 32. PL/I code for acquiring network diameter

The acquired diameter is stored in MAX_D and used in the termination control algorithm. Note that SYS$TRNLOG is the logical name translation routine in VMS.

16.6.2 ATTACHMENT OF THE TERMINATION CONTROL ALGORITHM

The attaching process is relatively straightforward. So we concentrate only on the functional description of the attached program rather than the programs that produce the attachment.

The current version of MODEL uses, as default, the Gauss-Seidel iterative procedure to solve simultaneous equations. The iterative procedure consists of three parts:

i) The initialization sequence,

ii) the convergence procedure, and

iii) the Gauss-Seidel recalculation loop.

Detailed description of the initialization sequence generation and convergence procedure generation can be found in [Greenberg, 81].

The initialization sequence should be modified if the module is involved in a DSE system. This is accomplished in the scheduling stage by modifying the initialization node list (see previous section). The termination control algorithm is attached to the recalculation loop which consists of a) controlled READ and unpacking, b) controlled WRITE, and c) token calculation.

The original recalculation loop has the following structure:

```
1. $ITER_CNVRG_1 = '0'B;
2. DO $ITER_CNTR_1 = 1 TO 50 WHILE(^$ITER_CNVRG_1);
3.    $ITER_CNVRG_1 = '1'B;
4.    ‹list of recalculations›
5.      ‹list of nested iterative procedures›
6.      IF ^$ITER_CNVRG_2 THEN $ITER_CNVRG_1 = '0'B;
7. END;
```

FIGURE 33. The recalculation procedure for simultaneous equations

Where $ITER_CNVRG_i and $ITER_CNTR_i are the Boolean and integer control variables of i-th nesting level respectively. The number 50 is the maximum iteration limit (taken from the BLOCK structure).

The attachment of the termination algorithm is determined by the existence of DEPENDS_ON statement in a simultaneous block. In other words, for each simultaneous block containing DEPENDS_ON statement(s), there is an extra outer block - the termination control condition.

After attaching the termination control algorithm, assuming MAX_D is the maximum diameter acquired from the JCL statement, the iterative procedure has the following structure:

```
$CNT$1=0; $PRE_ITO$1=1; $RLK$1='1'b;
DO WHILE($PRE_ITO$1 <= D + 1); /* Termination control condition */
1. $ITER_CNVRG_1 = '0'B;
2. DO $ITER_CNTR_1 = 1 TO 50
      WHILE (^$ITER_CNVRG_1 & $PRE_ITO$1<=MAX_D+1);
3.    $ITER_CNVRG_1= '1'B;
    <read label>:;
    <$RLK$ controlled READ action> /* Controlled READ      */
    <$RLK$ controlled unpacking>   /* Controlled Unpacking */
4.    <list of recalculations>
5.    <list of nested iterative procedures>
6.    IF ^$ITER_CNVRG_2 THEN $ITER_CNVRG_1 = '0'B;
    <$RLK$ controlled unpacking for control field $ITS>
    IF $CNT$1=0 THEN $ITO$1=0;     /* Token calculation    */
    ELSE DO;
      IF ^$ITER_CNVRG_1 THEN $ITO$1=0;
      ELSE IF $PRE_ITO$ < MIN_INP(1) THEN $1TO$1=$PRE_ITO$1 + 1;
                                     ELSE $ITO$1=MIN_INP1(1)+ 1;
    END /* $CNT$1>0 */;
    $PRE_ITO$1=$ITO$1;
    CALL WRITE_IT1;                /* Write tokens         */
    IF $PRE_ITO$1 < MAX_D + 1
    THEN DO;                       /* Controlled WRITE     */
      <output actions in the iterative procedure>
    END /* OF $CNT$1<= */;
    $RLK$1='0'B;                   /* Unlock $RLK$         */
7. END /* $ITER_CNTR_1 */ ;
END /* $CNT$1<=$PRE_ITO$1 */ ;
```

FIGURE 34. The recalculation procedure with termination control

The correspondence between the attached version and the original version can be identified by the statement numbers marked.

The algorithm uses several control variables, the correspondence with the variable names given in the termination algorithm description is as follows.

```
$CNT$i      ------ corresponds to the index of the i-th block
                   calculation (unused, for debugging)
$PRE_ITO$i  ------ corresponds to the i-th block's PRE_O variable
$ITO$i      ------ corresponds to the i-th block's O variable
$RLK$i      ------ the i-th block's control variable for controlling
                   the execution of the first READ operation
```

RLKi is used to disable the first READ operation inside of the i-th iterative block associated with a DSE system. The starting values of the equations must be from the INITIAL statements.

There are also two procedures used by the control algorithm: MIN_INPi and WRITE_ITi for each iterative block i. MIN_INPi corresponds to the MIN function in the description of the control

algorithm. It is generated by the use of the SIMU_NODES structure. Assuming FI1,FI2,...FIn are the input records to the module involved in a DSE system at the i-th iterative block, MIN_INPi has the following format in PL/I:

```
MIN_INPi:PROC(I) RETURNS(FIXED BIN);
DCL (I,MIN_I) FIXED BIN;
MIN_I=FI1.$IT$;
IF FI2.$IT$ < MIN_I THEN MIN_I=FI2.$IT$;
        :
IF FIn.$IT$ < MIN_I THEN MIN_I=FIn.$IT$;
RETURN(MIN_I);
END MIN_INPi;
```

WRITE_ITi is a procedure for sending the control tokens. It is also generated by using the SIMU_NODE structure. Assuming FO1, FO2, ... FOn are the output records involved in a DES system at i-th block, The WRITE_ITi procedure has the following format:

```
WRITE_ITi:PROC;
FO1.$IT$ = $ITO$i;
FO2.$IT$ = $ITO$i;
    :
FOn.$IT$ = $ITO$i;
RETURN;
END WRITE_ITi;
```

APPENDIX A

EXAMPLES

A1. THE RESOURCE ALLOCATION EXAMPLE

This appendix provides the details omitted in the previous parts. In reading this appendix, some basic knowledge of VAX-11 PL/I and VMS is necessary in order to justify the correctness of the implementation and consistency with the description given before.

A1.1 THE PHILOSOPHER MODULE

```
MODEL PROCESSOR: VERSION WITH BLOCK STRUCTURE ON VAX 11/750    OCTOBER 04, 1984      10:17:32.90

/*****************************************************************************/
/*                        P1 MODULE SPECIFICATION                            */
/*****************************************************************************/

   1     MODULE:P1:
   2     SOURCE:ALLOC1:
   3     TARGET:REQREL1:
   4

/*****************************************************************************/
/*                          FILE DESCRIPTIONS:                               */
/*****************************************************************************/


/*****************************************************************************/
/*                    DESCRIPTION OF ALLOC1     FILE                         */
/*****************************************************************************/

   4     1 ALLOC1 FILE ORG MAIL,        /* FILE OF ALLOCATION */
   4       2 MSGA(*) IS RECORD,             /* INDIVIDUAL ALLOCATION     */
   4         3 PROC_ID IS FLD(CHAR 11),     /* RECEIVING PROCESS         */
   4         3 CLOCKA  IS FLD(PIC '999999999'1);   /* CLOCK AT ALLOCATION      */
```

```
/*********************************************************************/
/*                  DESCRIPTION OF REQREL1    FILE                   */
/*********************************************************************/
   5     1 REQREL1 FILE ORG IS MAIL.          /* TARGET FILE OF REQ_REL    */
   5       2 MSGR(*) IS RECORD.               /* INDIVIDUAL REQ_REL        */
   5         3 PROC_ID  IS FLD(PIC '9').      /* SENDING PROCESS           */
   5         3 RQ_OR_RL IS FLD(CHAR 3).       /* REQUEST=REQ, RELEASE=REL  */
   5         3 CLOCKR   IS FLD(PIC '9999999999'), /* CLOCK AT REQ_REL      */
   5         3 RES(5)   IS FLD(PIC '9');      /* NEEDED RESOURCES VECTOR   */
   6
   6       (I1,J) SUBSCRIPT;
   7
   7       IX1 FLD (NUM(5));
   8
   8       IX1(I1)=IF I1=1 THEN 1
   9              ELSE IF RQ_OR_RL(I1)='REQ' THEN IX1(I1-1)+1
   8                                         ELSE IX1(I1-1);
   9
   9     /* ------------------------ PHIL 1 ------------------------ */
   9       /* LIFE TIME OF P1 */
   9       END.REQREL1.MSGR(I1)=REQREL1.RQ_OR_RL(I1)='REL' & I1>10;
  10
  10       REQREL1.CLOCKR(I1)=IF I1=1 THEN TIME
  10                          ELSE ALLOC1.CLOCKA(IX1(I1-1))-ALLOC1.CLOCKA(IX1(I1-1))
  10                         +TIME;
  11
  11       REQREL1.PROC_ID(I1)=1;
  12
  12       REQREL1.RQ_OR_RL(I1)=IF I1=1 THEN  'REQ'   /* REQUESTING */
  12                            ELSE IF REQREL1.RQ_OR_RL(I1-1)='REQ'
  12                                 THEN 'REL'
  12                                 ELSE 'REQ';
  13
  13       REQREL1.RES(I1,J)= (J=1 ! J=2);   /* (J=MOD(K,5) ! J=MOD(K+1,5)); */
  14
  14       $YSGEN1 IS GROUP(INTERIM,IX1(*));
  15       $YSGEN2 IS GROUP(END.REQREL1.MSGR(*));
```

RANGE TABLE

RANGE NO.	RANGE DEFINITION	WHERE DEFINED
1	END OF FILE	ALLOC1.MSGA
2	END	REDREL1.MSGR
3	CONSTANT LIMIT:5	REDREL1.RES

	RANGE NO.		
	1	2	3
NODE NAME	DIMENSION NO.		
*ASSERTION(S):			
AASS100	1V		
AASS110	1V		
AASS120	1V		
AASS130	1V	2V	
AASS80	1V		
AASS90	1V		
*Q_NAME:(ALLOC1.)			
CLOCKA	1V2		
MSGA	1V		
PROC_ID	1V		
*Q_NAME:(END.REDREL1.)			
MSGR	1V2		
*Q_NAME:(INTERIM.)			
IX1	1V2		
*Q_NAME:(REDREL1.)			
CLOCKR	1V		
MSGR	1V		
PROC_ID	1V		
RES	1V	2P	
RQ_OR_RL	1V2		
*GLOBAL SUBSCRIPT:			
I1	1V		
J		1V	

NOTE: ENTRY COL.1-DIMENSION NUMBER
2-PHYSICAL(P)/VIRTUAL(V) DIMENSION
3-WINDOW SIZE. IF MORE THEN ONE

FLOWCHART REPORT

CODE$	NAME	DESCRIPTION	EVENT
====	P1	MODULE NAME	PROCEDURE HEADING
37	ALLOC1	FILE	OPEN FILE
0		ITERATION	FOR $I1 UNTIL END_F SPECIFIED
33	AASS120	ASSERTION	
47	REQREL1.RQ_OR_RL	FIELD IN RECORD REQREL1.MSGR	TARGET OF ASSERTION: AASS120
35	AASS80	ASSERTION	
50	INTERIM.IX1	FIELD	TARGET OF ASSERTION: AASS80
36	AASS90	ASSERTION	
51	END.REQREL1.MSGR	SPECIAL NAME	TARGET OF ASSERTION: AASS90
0		ITERATION	FOR $I2 UNTIL CONSTANT LIMIT
34	AASS130	ASSERTION	
49	REQREL1.RES	FIELD IN RECORD REQREL1.MSGR	TARGET OF ASSERTION: AASS130
		END ITERATION	FOR $I2
32	AASS110	ASSERTION	
46	REQREL1.PROC_ID	FIELD IN RECORD REQREL1.MSGR	TARGET OF ASSERTION: AASS110
31	AASS100	ASSERTION	
48	REQREL1.CLOCKR	FIELD IN RECORD REQREL1.MSGR	TARGET OF ASSERTION: AASS100
45	REQREL1.MSGR	RECORD IN FILE REQREL1	WRITE RECORD
		CONDITIONAL BLOCK	SUBLINEAR RANGE IS V*A INTERIM.IX1
38	ALLOC1.MSGA	RECORD IN FILE ALLOC1	READ RECORD
39	ALLOC1.PROC_ID	FIELD IN RECORD ALLOC1.MSGA	
40	ALLOC1.CLOCKA	FIELD IN RECORD ALLOC1.MSGA	
		END CONDITIONAL BLOCK	FOR INTERIM.IX1
		END ITERATION	FOR $I1
44	REQREL1	FILE	CLOSE FILE
52	$YSGEN1	GROUP	
53	$YSGEN2	GROUP	
		END	

--- PL/I PROGRAM ---

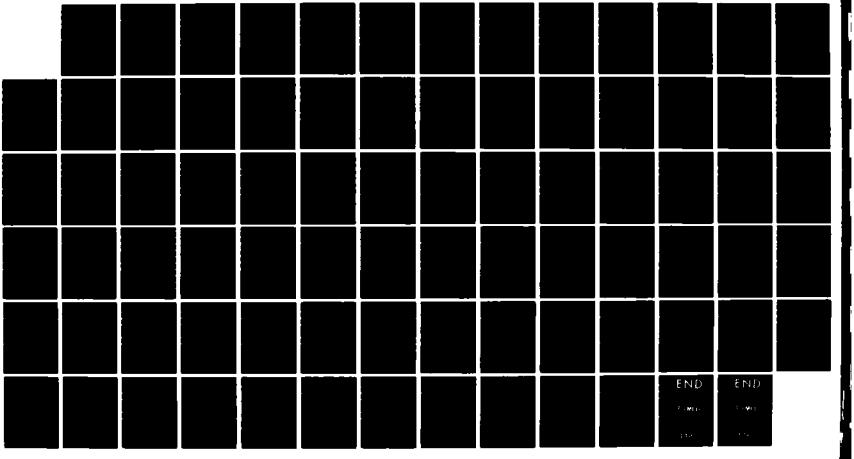
```
P1: PROCEDURE OPTIONS(MAIN);
DCL ALLOC1S RECORD SEQL INPUT;
DCL $FSTALLOC1S BIT(1) INIT('1'B);
DCL EXIST_ALLOC1S BIT(1) INIT('1'B);
DCL ENDFILE$ALLOC1S BIT(1) INIT('0'B);
DCL ALLOC1_S CHAR(20) VARYING INIT('');
DCL ALLOC1_INDX FIXED BIN;
DCL $B_INTERIM$IX1 BIT(1), $R_INTERIM$IX1 FIXED BIN;
DCL REQREL1_MSGR_S CHAR(19) VARYING;
DCL REQREL1_MSGR_S_F CHAR(19);
DCL REQREL1_MSGR_SC BIT(152) BASED(ADDR(REQREL1_MSGR_S_F));
DCL REQREL1_MSGR_INDX FIXED BIN;
DCL $X4 FIXED BIN;
DCL ALLOC1_MSGA_S CHAR(20) VARYING;
DCL ALLOC1_MSGA_INDX FIXED BIN;
DCL REQRELIT RECORD SEQL OUTPUT;
DCL $FSTREQRELIT BIT(1) INIT('1'B);
DCL $ERROR_BUF CHAR(270) VAR;
DCL ERRORF FILE RECORD OUTPUT;
DCL ERRORF_BIT BIT(1) STATIC INIT('1'B);
DCL $NOT_DONE(20) BIT(1);
DCL $ERROR BIT(1) INIT('0'B);
DCL $TMP_VAL FLOAT BIN;
DCL /$RDLF$.$R_L1 LABEL;
DECLARE
    1 ALLOC1,
     3 MSGA,
      5 PROC_ID CHAR(11),
      5 CLOCKA(2) PIC '999999999',
```

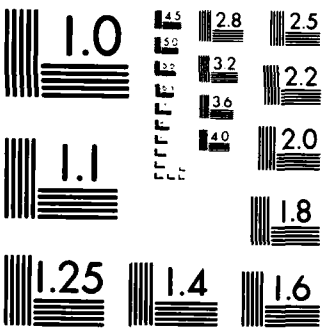
```
DECLARE
   1 REQREL1,
    2 MSGR,
     3 PROC_ID PIC'9',
     3 RQ_OR_RL(2) CHAR(3),
     3 CLOCKR PIC'9999999999',
     3 RES(5) PIC'9';
DECLARE
   1 INTERIM,
    2 $YSGEN1,
     3 IX1(2) PIC'99999',
    2 $YSGEN2,
     3 END$REQREL1_MSGR(2) BIT(1) ;
DCL $I1 FIXED BIN;
DCL $I2 FIXED BIN;
DCL (TRUE,SELECTED) BIT(1) INIT('1'B);
DCL (FALSE,NOT_SELE,NOT_SELECTED) BIT(1) INIT('0'B);
DCL TIME BUILTIN;
ON ENDFILE(ALLOC1S) BEGIN;
$RD_LP$:
 END;
ON UNDEFINEDFILE(ERRORF)  ERRORF_BIT='0'B;
DECLARE PLI$_CNVERR GLOBALREF VALUE FIXED BIN(31);
DECLARE RMS$_RLK GLOBALREF VALUE FIXED BIN(31);
ON ERROR BEGIN;
IF ONCODE()=RMS$_RLK THEN GOTO $RD_LP$;
IF ^(SERROR) THEN CALL RESIGNAL();
IF ONCODE()=PLI$_CNVERR THEN DO;
$ERROR='0'B;
IF ERRORF_BIT THEN WRITE FILE(ERRORF) FROM (SERROR_BUF);
 END;
ELSE CALL RESIGNAL();
 END;
MAILBOX_NAME='ALLOC1S_MBX';
MAX_LENGTH=20;
STS$VALUE=SYS$CREMBX(PERMANENT,CHANNEL,MAX_LENGTH,,PROT_MASK,,MAILBOX_NAME);
LN_ALLOC1S='ALLOC1S';
EQV_NAME='';
STS$VALUE=SYS$TRNLOG(LN_ALLOC1S,L_LENGTH,EQV_NAME,,,);
if ^sts$success then put skip list('translation error');
OPEN FILE(ALLOC1S) INPUT TITLE(EQV_NAME);
$X$=SUBSTR(EQV_NAME,1,L_LENGTH);
STS$VALUE=SYS$ASSIGN($X$,CHANNEL,,,);
IF ^STS$SUCCESS THEN EXIST_ALLOC1S='0'B;
$I1 =0;
$NOT_DONE(1)='1'B;
DO  WHILE($NOT_DONE(1));
  $I1 = $I1 +1;
  IF $I1=1 THEN REQREL1.RQ_OR_RL(2)='REQ';
   ELSE IF REQREL1.RQ_OR_RL(1)='REQ' THEN REQREL1.RQ_OR_RL(2)='REL';
   ELSE REQREL1.RQ_OR_RL(2)='REQ';
  IF $I1=1 THEN INTERIM.IX1(2)=1;
   ELSE IF REQREL1.RQ_OR_RL(2)='REQ' THEN INTERIM.IX1(2)=INTERIM.IX1(1)+1;
   ELSE INTERIM.IX1(2)=INTERIM.IX1(1);
  IF $I1=1 THEN DO;
  IF INTERIM.IX1(2)=1 THEN DO;
   $B_INTERIM$IX1=1'B;
   $R_INTERIM$IX1=0;
   END;
  ELSE DO;
   $B_INTERIM$IX1='0'B;
   $R_INTERIM$IX1=1;
   END;
END;
 ELSE IF (INTERIM.IX1(2)=INTERIM.IX1(1)) THEN DO;
 $B_INTERIM$IX1='1'B;
 $R_INTERIM$IX1=0;
 END;
```

```
 ELSE DO:
 $B_INTERIM$IX1= '0'B:
 $R_INTERIM$IX1=1:
 END:
END$REQREL1_MSGR(2)=REQREL1.RQ_OR_RL(2)='REL'&$I1:
DO $I2 =1 TO 5:
  REQREL1.RES($I2)=$I2=1!$I2=2:
END:
REQREL1.PROC_ID=1:
IF $I1=1 THEN REQREL1.CLOCKR=TIME:
 ELSE REQREL1.CLOCKR=ALLOC1.CLOCKA(1-INTERIM.IX1(1)+INTERIM.IX1(1)-
ALLOC1.CLOCKA(1-INTERIM.IX1(1)+INTERIM.IX1(1)+TIME:
REQREL1_MSGR_INDX=1:
SUBSTR(REQREL1_MSGR_SC,REQREL1_MSGR_INDX*8-7,1*8)=UNSPEC(REQREL1.PROC_ID) :
REQREL1_MSGR_INDX=REQREL1_MSGR_INDX+1 :
SUBSTR(REQREL1_MSGR_S_F,REQREL1_MSGR_INDX,3)=REQREL1.RQ_OR_RL(2) :
REQREL1_MSGR_INDX=REQREL1_MSGR_INDX+3 :
SUBSTR(REQREL1_MSGR_SC,REQREL1_MSGR_INDX*8-7,1*8)=UNSPEC(REQREL1.CLOCKR) :
REQREL1_MSGR_INDX=REQREL1_MSGR_INDX+10 :
DO $I2 =1 TO 5:
  SUBSTR(REQREL1_MSGR_SC,REQREL1_MSGR_INDX*8-7,1*8)=UNSPEC(REQREL1.RES($I2)) :
  REQREL1_MSGR_INDX=REQREL1_MSGR_INDX+1 :
END:
REQREL1_MSGR_S=SUBSTR(REQREL1_MSGR_S_F,1,REQREL1_MSGR_INDX-1):
LN_REQREL1T='REQREL1T':
STS$VALUE=SYS$TRNLOG(LN_REQREL1T,L_LENGTH,EQV_NAME,...):
IF ^STS$SUCCESS THEN PUT SKIP LIST('TRANSLATION ERROR'):
$X$=SUBSTR(EQV_NAME,1,L_LENGTH):
STS$VALUE=SYS$ASSIGN($X$,CHANNEL,...):
IF ^STS$SUCCESS THEN DO:
PUT SKIP LIST('**ERROR: NO CHANNEL ASSIGNED(MAIL)'):
WRITE FILE(REQREL1T) FROM (REQREL1_MSGR_S):
 END:
 ELSE DO:
 STS$VALUE=SYS$QIO(1,CHANNEL,IO$_WRITEVBLK+IO$M_NOW,IO_STATUS,...
ADDR(REQREL1_MSGR_S_F),LENGTH(REQREL1_MSGR_S),...):
 IF ^STS$SUCCESS THEN PUT SKIP LIST('WRITE QIO UNSUCCESSFUL' CHECK REC. SIZE'):
 STS$VALUE=SYS$WAITFR(1):
 STS$VALUE=SYS$DASSGN(CHANNEL):
   END:
IF $B_INTERIM$IX1 THEN
DO :
  $X4 = INTERIM.IX1(2):
  $RD_ALLOC1S:
  $R_L=$RD_ALLOC1S:
    READ FILE(ALLOC1S) INTO (ALLOC1_MSGA_S):
  ALLOC1_MSGA_INDX=1:
  $ERROR_BUF=ALLOC1_MSGA_S:
  ALLOC1_MSGA_S=ALLOC1_MSGA_S(copy(' ',20):
  ALLOC1.PROC_ID=SUBSTR(ALLOC1_MSGA_S,ALLOC1_MSGA_INDX,1) :
  ALLOC1_MSGA_INDX=ALLOC1_MSGA_INDX+1 :
  $ERROR='1'B :
  UNSPEC(ALLOC1.CLOCKA(2))=UNSPEC(SUBSTR(ALLOC1_MSGA_S,ALLOC1_MSGA_INDX,9)) :
  ALLOC1.CLOCKA(2) = ALLOC1.CLOCKA(2) :
  IF $ERROR THEN $ERROR= '0'B :
  ALLOC1_MSGA_INDX=ALLOC1_MSGA_INDX+9 :
END:
IF END$REQREL1_MSGR(2) THEN $NOT_DONE(1)='0'B:
END$REQREL1_MSGR(1) = END$REQREL1_MSGR(2):
INTERIM.IX1(1) = INTERIM.IX1(2):
REQREL1.RQ_OR_RL(1) = REQREL1.RQ_OR_RL(2) :
IF $B_INTERIM$IX1 THEN ALLOC1.CLOCKA(1) = ALLOC1.CLOCKA(2) :
E:
CLOSE FILE(REQREL1T):
RETURN:
END P1:
```

AD-A150 009 VERY-HIGH LEVEL CONCURRENT PROGRAMMING(U) MOORE SCHOOL OF ELECTRICAL ENGINEERING PHILADELPHIA PA DEPT OF COMPUTER AND INFORMATION SCIENCES Y SHI DEC 84 3/3

UNCLASSIFIED N00014-83-K-0560 F/G 9/2 NL





MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS 1963 A

A1.2 THE R MODULE

```
MODEL PROCESSOR: VERSION WITH BLOCK STRUCTURE ON VAX 11/750    OCTOBER 04, 1984      10:31:36.34

/*:*****************************************************************/
/*                        R MODULE SPECIFICATION                     */
/*:*****************************************************************/

  1     MODULE:R:
  2     SOURCE: REQREL:
  3     TARGET: ALLOC,QUEUE,SIMULATION:
  4             /* THE FILE QUEUE, SIMULATION ARE FOR REPORTING */
  4
  4       /* THE MAIL FILE RECEIVING MESSAGES FROM PHILOSOPHERS */

/*****************************************************************/
/*                        FILE DESCRIPTIONS:                         */
/*****************************************************************/


/*****************************************************************/
/*                  DESCRIPTION OF REQREL     FILE                  */
/*****************************************************************/

  4       1 REQREL IS FILE ORG IS MAIL,
  4        2 MSGRM(*) IS RECORD,
  4          3 PROC_IDM IS FIELD (PIC'9'), /* ID OF PROCESS*/
  4          3 RQ_OR_RLM IS FIELD (CHAR 3),/* REQUEST=REQ,RELEASE=REL */
  4          3 CLOCKRM IS FIELD (PIC'9999999999'), /*TIME OF MESSAGE*/
  4          3 RESM(5) IS FIELD (PIC'9'):  /*VECTOR OF RESOURCES*/
  5
  5       /* THE LIFE TIME OF R */
  5       END,REQREL,MSGRM(I)=DATE>='840923':
  6
  6       /* THE DISTRIBUTING FILE _NDING MESSAGES TO PHILOSOPHERS */

/*****************************************************************/
/*                  DESCRIPTION OF ALLOC      FILE                  */
/*****************************************************************/

  6       1 ALLOC IS FILE KEY IS PROC_ID ORG IS POST,
  6        2 MSGAS(*) IS GRP,/*GROUP OF ALLOCATION MESSAGES*/
  6          3 MSGA(0:99) IS RECORD,/*INDIVIDUAL MESSAGE*/
  6           4 PROC_ID IS FIELD (CHAR (1),/*ALLOCATED PROCESS*/
  6           4 CLOCKA IS FIELD (PIC'9999999999'):/*TIME OF ALLOCATION*/
  7
  7       /* DEFINE THE # OF THE OUTPUT RECORDS CAN BE DISTRIBUTED AT THE ITH CYCLE */
  7       SIZE,ALLOC,MSGA(I)= IF SIZE,PROC(I)>0 THEN  OUT_IY(I,SIZE,PROC(I)) ELSE 0:
  8

/*****************************************************************/
/*                  DESCRIPTION OF SIMULATI   FILE                  */
/*****************************************************************/

  8       1  SIMULATION  IS FILE,
  8          2 HD1 IS RECORD,
  8            3 HDF1 IS FIELD(CHAR 125),
  8          2 HD2 IS RECORD,
```

```
 8           3 HDF2 IS FIELD(CHAR 125),
 8         2 HD3 IS RECORD,
 8           3 HDF3 IS FIELD(CHAR 125),
 8         2 EVENT(*) IS RECORD,
 8           3 REQUEST IS GRP,
 8             4 PROC_IDM  IS FIELD(CHAR 4),
 8             4 FILLER1   IS FIELD(CHAR 1),
 8             4 RQ_OR_RLM IS FIELD(CHAR 3),
 8             4 FILLER2   IS FIELD(CHAR 1),
 8             4 RESM(5)   IS FIELD(PIC '9'),
 8             4 FILLER3   IS FIELD(CHAR 1),
 8             4 CLOCKRM   IS FIELD(PIC 'B(12)9'),
 8             4 FILLER4   IS FIELD(CHAR 2),
 8           3 ALLOCATION(*) IS GRP,
 8             4 FILLER5   IS FIELD(CHAR 1),
 8             4 PROC_IDA  IS FIELD(CHAR 4),
 8             4 FILLER6   IS FIELD(CHAR 1),
 8             4 CLOCKA    IS FIELD(PIC 'B(12)9');
 9
 9
 9       SIMULATION.HDF1='          REQUEST                                    ALLOCATION';
10       SIMULATION.HDF2='P_ID R/L RESRC     TIME        P_ID     TIME      ||
10                                                  P_ID     TIME      ||
10                                                ' P_ID     TIME';
11       SIMULATION.HDF3='==== === ===== ============   ==== ============||
11                                                ' ==== ============||
11                                                ' ==== ============';
12       (FILLER1,FILLER2,FILLER3,FILLER4,FILLER5,FILLER6) = ' ';
13       SIMULATION.PROC_IDM =REQREL.PROC_IDM;
14       SIMULATION.RESM     =REQREL.RESM;
15       SIMULATION.CLOCKRM  =REQREL.CLOCKRM;
16       SIMULATION.PROC_IDA =SUBSTR(ALLOC.PROC_ID,6,1);
17       SIMULATION.CLOCKA   =ALLOC.CLOCKA;
18
18       /* THE QUEUE FILE, DEFINED AS TARGET TO CHECK THE R */

/*********************************************************************/

/*                  DESCRIPTION OF QUEUE      FILE                   */

/*********************************************************************/

18       1 QUEUE IS FILE,
18        2 PROC_Q(*) IS RECORD, /* RECORDING THE HISTORY OF THE QUEUE */
18         3 PROC(0:99) IS GROUP,/* PROCESS QUEUE FOR EACH I */
18           4 PROC_ID IS FIELD (PIC'9'),/* ID OF PROCESS */
18           4 IN_IX IS FIELD (PIC'999999999'),
18                        /*INDEX OF PROCESS IN PRECEEDING QUEUE*/
18           4 OUT_IX IS FIELD (PIC'9'),/* INDEX OF PROCESS IN ALLOCATIONS*/
18           4 RES(5) IS GROUP,         /* RESOURCE VECTOR */
18            5 CLAIM IS FIELD (PIC'9'),/*MAXIMUM RESOURCES CLAIMED BY PROCESS*/
18            5 SUM_REQ IS FIELD (PIC'9'),/*SUMS OF REQUESTS FOR RESOURCES IN
18                                          QUEUE ORDER */
18            5 SAT IS FIELD (BIT(1));/*WHETHER RESOURCES ARE AVAILABLE TO,
19                                     SATISFY CLAIMS IN ORDER OF RESOURCES*/
19       1 RES_LIMIT IS GROUP,
19        2 NUM_RES(5) IS FIELD (PIC'9');/*NUMBER OF RESOURCES IN SYSTEM THAT
20                                        MAY BE ALLOCATED*/
20       /*DATA PARAMETERS*/
20       (I,J,K) ARE SUBSCRIPTS;/*I: SUBSCRIPT OF REQUEST/RELEASE MESSAGES;
21                               J: SUBSCRIPT OF RESOURCES;
21                               K: SUBSCRIPT OR PROCESSES IN QUEUE. */
21
21       /* DEFINE SIZE OF QUEUE. */
21       SIZE.PROC(I)=IF I=1
21                    THEN 1
21                    ELSE IF REQREL.RQ_OR_RLM(I)='REL'
21                         THEN SIZE.PROC(I-1)-1
21                         ELSE SIZE.PROC(I-1)+1;
```

```
22
23
22      /* DEFINE THE NUMBER OF AVAILABEL RESOURCES IN THE SYSTEM. */
22      NUM_RES(J)=1: /*ONE FORK IN EACH POSITION*/
23
23      /* DEFINE THE QUEUE CONTENTS: */
23      QUEUE.PROC_ID(I,K)= IF REQREL.RQ_OR_RLM(I)='REQ' & (K=SIZE.PROC(I))
23                          THEN REQREL.PROC_IDM(I) /* PUSH */
23                          ELSE QUEUE.PROC_ID(I-1,IN_IX(I,K)): /* COPY */
24
24      /* DEFINE THE INDEX FOR VARIABLELY SIZED QUEUE. */
24      IN_IX(I,K)= IF REQREL.RQ_OR_RLM(I)='REQ'
24                  THEN K
24                  ELSE IF REQREL.PROC_IDM(I)^=QUEUE.PROC_ID(I-1,K)
24                       THEN IF K=1
24                            THEN 1
24                            ELSE IN_IX(I,K-1)+1
24                       ELSE IF K=1  /* MUST BE REL. THE Q SIZE-1 BUT PRE.INX+1 */
24                            THEN 2
24                            ELSE IN_IX(I,K-1)+2:
25
25      /* DEFINE THE OUTPUT INDIRECT INDEX */
25      OUT_IX(I,K)=IF REQREL.RQ_OR_RLM(I)='REL'
25                  THEN IF SAT(I,K,5)& ^SAT(I-1,IN_IX(I,K),5)
25                       THEN IF K=1
25                            THEN 1
25                            ELSE OUT_IX(I,K-1)+1
25                       ELSE IF K=1
25                            THEN 0
25                            ELSE OUT_IX(I,K-1)
25                  ELSE IF K=SIZE.PROC(I)&SAT(I,K,5)
25                       THEN 1
25                       ELSE 0:
26
26      /* DEFINE CALIM FOR EACH PROCESS. */
26      CLAIM(I,K,J)= IF REQREL.RQ_OR_RLM(I)='REQ' & (K=SIZE.PROC(I))
26                    THEN REQREL.RESM(I,J)
26                    ELSE CLAIM(I-1,IN_IX(I,K),J):
27      SUM_REQ(I,K,J)= IF K=1
27                      THEN QUEUE.CLAIM(I,K,J)
27                      ELSE QUEUE.CLAIM(I,K,J)+SUM_REQ(I,K-1,J):
28
28      /* TEST VARIABLE FOR DETERMINING THE ALLOCATABLE RESOURCES. */
28      SAT(I,K,J)=IF J=1
28                 THEN (SUM_REQ(I,K,J)<=NUM_RES(J) ! QUEUE.CLAIM(I,K,J)=0)
28                 ELSE SAT(I,K,J-1) &
28                      (SUM_REQ(I,K,J)<=NUM_RES(J) ! QUEUE.CLAIM(I,K,J)=0):
29
29
29      /*       EQUATIONS FOR VARIABLES IN FILE ALLOCA   */
29
29      /* DEFINE THE MAILBOX ADDRESS */
29      ALLOC.PROC_ID(I,OUT_IX(I,K))=
29              IF (K=1 & OUT_IX(I,K)=1)!(K>1)&(OUT_IX(I,K)^=OUT_IX(I,K-1))
29              THEN CHPTRLB('ALLOC'!!LEFT.J(QUEUE.PROC_ID(I,K))!!'S_MBX'):
30      /* DEFINE ALLOCATION TIME */
30      ALLOC.CLOCKA(I,OUT_IX(I,K))=IF (K=1 &OUT_IX(I,K)=1)
30                                  !(K>1 & OUT_IX(I,K)>OUT_IX(I,K-1))
30                                  THEN REQREL.CLOCKRM(I):
31
31      SIMULATI.RQ_OR_RL=REQREL.RQ_OR_RL:
32        $YSGEN1 IS GROUP(END,REQREL.MSGRM(*)):
33        $YSGEN2 IS GROUP(SIZE.QUEUE.PROC(*)):
34        $YSGEN3 IS GROUP(SIZE.ALLOC.MSGA(*)):
```

FLOWCHART REPORT

	$	NAME	DESCRIPTION	EVENT
	R		MODULE NAME	PROCEDURE HEADING
	76	REQREL	FILE	OPEN FILE
	56	AASS90	ASSERTION	
	84	SIMULATI.HDF1	FIELD IN RECORD SIMULATI.HD1	TARGET OF ASSERTION: AASS90
	83	SIMULATI.HD1	RECORD IN FILE SIMULATI	WRITE RECORD
0			ITERATION	FOR $I1 UNTIL CONSTANT LIMITS
	45	AASS220	ASSERTION	
	105	INTERIM.NUM_RES	FIELD	TARGET OF ASSERTION: AASS220
			END ITERATION	FOR $I1
	104	INTERIM.RES_LIMI	GROUP	
	32	AASS110	ASSERTION	
	31	AASS100	ASSERTION	
	86	SIMULATI.HDF2	FIELD IN RECORD SIMULATI.HD2	TARGET OF ASSERTION: AASS100
	85	SIMULATI.HD2	RECORD IN FILE SIMULATI	WRITE RECORD
	88	SIMULATI.HDF3	FIELD IN RECORD SIMULATI.HD3	TARGET OF ASSERTION: AASS110
	87	SIMULATI.HD3	RECORD IN FILE SIMULATI	WRITE RECORD
0			ITERATION	FOR $I1 UNTIL END.X SPECIFIED
	54	AASS50	ASSERTION	
	106	END.REQREL.MSGRM	SPECIAL NAME	TARGET OF ASSERTION: AASS50
	77	REQREL.MSGRM	RECORD IN FILE REQREL	READ RECORD
	78	REQREL.PROC_IDM	FIELD IN RECORD REQREL.MSGRM	
	39	AASS130	ASSERTION	
	91	SIMULATI.PROC_IDM	FIELD IN RECORD SIMULATI.EVENT	TARGET OF ASSERTION: AASS130
	79	REQREL.RQ_OR_RL	FIELD IN RECORD REQREL.MSGRM	
	109	AASS310	ASSERTION	
	93	SIMULATI.RQ_OR_RL	FIELD IN RECORD SIMULATI.EVENT	TARGET OF ASSERTION: AASS310
	44	AASS210	ASSERTION	
	107	SIZE.QUEUE.PROC	SPECIAL NAME	TARGET OF ASSERTION: AASS210
0			ITERATION	FOR $I2 UNTIL SIZE.X SPECIFIED
	47	AASS240	ASSERTION	
	70	QUEUE.IN_IX	FIELD IN RECORD QUEUE.PROC_Q	TARGET OF ASSERTION: AASS240
	46	AASS230	ASSERTION	
	69	QUEUE.PROC_ID	FIELD IN RECORD QUEUE.PROC_Q	TARGET OF ASSERTION: AASS230
			END ITERATION	FOR $I2
	80	REQREL.CLOCKRM	FIELD IN RECORD REQREL.MSGRM	
	41	AASS150	ASSERTION	
	97	SIMULATI.CLOCKRM	FIELD IN RECORD SIMULATI.EVENT	TARGET OF ASSERTION: AASS150
0			ITERATION	FOR $I2 UNTIL CONSTANT LIMITS
	81	REQREL.RESM	FIELD IN RECORD REQREL.MSGRM	
	40	AASS140	ASSERTION	
	95	SIMULATI.RESM	FIELD IN RECORD SIMULATI.EVENT	TARGET OF ASSERTION: AASS140
0			ITERATION	FOR $I2 UNTIL SIZE.X SPECIFIED
	49	AASS260	ASSERTION	
	73	QUEUE.CLAIM	FIELD IN RECORD QUEUE.PROC_Q	TARGET OF ASSERTION: AASS260
	50	AASS270	ASSERTION	
	74	QUEUE.SUM_REQ	FIELD IN RECORD QUEUE.PROC_Q	TARGET OF ASSERTION: AASS270
	51	AASS280	ASSERTION	
	75	QUEUE.SAT	FIELD IN RECORD QUEUE.PROC_Q	TARGET OF ASSERTION: AASS280
	72	QUEUE.RES	GROUP IN RECORD QUEUE.PROC_Q	
			END ITERATION	FOR $I3
			END ITERATION	FOR $I2
			ITERATION	FOR $I2 UNTIL SIZE.X SPECIFIED
0			ASSERTION	
	48	AASS250	FIELD IN RECORD QUEUE.PROC_Q	TARGET OF ASSERTION: AASS250
	71	QUEUE.OUT_IX	ASSERTION	
	53	AASS300	ASSERTION	
	52	AASS290	GROUP IN RECORD QUEUE.PROC_Q	
	68	QUEUE.PROC	END ITERATION	FOR $I2
nf				
	57	AASS70	ASSERTION	
	108	SIZE.ALLOC.MSGA	SPECIAL NAME	TARGET OF ASSERTION: AASS70
0			ITERATION	FOR $I2 UNTIL SIZE.X SPECIFIED
	61	ALLOC.CLOCKA	FIELD IN RECORD ALLOC.MSGA	TARGET OF ASSERTION: AASS300
	43	AASS170	ASSERTION	
	103	SIMULATI.CLOCKA	FIELD IN RECORD SIMULATI.EVENT	TARGET OF ASSERTION: AASS170

```
60   ALLOC.PROC_ID          FIELD IN RECORD ALLOC.MSGA        TARGET OF ASSERTION: AASS2
42   AASS160                ASSERTION
101  SIMULATI.PROC_IDA      FIELD IN RECORD SIMULATI.EVENT    TARGET OF ASSERTION: AASS160
59   ALLOC.MSGA             RECORD IN FILE ALLOC              WRITE RECORD
34   AASS120AB              ASSERTION
100  SIMULATI.FILLER5       FIELD IN RECORD SIMULATI.EVENT    TARGET OF ASSERTION: AASS120AB
33   AASS120                ASSERTION
102  SIMULATI.FILLER6       FIELD IN RECORD SIMULATI.EVENT    TARGET OF ASSERTION: AASS120
99   SIMULATI.ALLOCATI      GROUP IN RECORD SIMULATI.EVENT
                            END ITERATION                     FOR $I2
58   ALLOC.MSGAS            GROUP IN FILE ALLOC
67   QUEUE.PROC_Q           RECORD IN FILE QUEUE              WRITE RECORD
38   AASS120AF              ASSERTION
92   SIMULATI.FILLER1       FIELD IN RECORD SIMULATI.EVENT    TARGET OF ASSERTION: AASS120AF
37   AASS120AE              ASSERTION
94   SIMULATI.FILLER2       FIELD IN RECORD SIMULATI.EVENT    TARGET OF ASSERTION: AASS120AE
36   AASS120AD              ASSERTION
96   SIMULATI.FILLER3       FIELD IN RECORD SIMULATI.EVENT    TARGET OF ASSERTION: AASS120AD
35   AASS120AC              ASSERTION
98   SIMULATI.FILLER4       FIELD IN RECORD SIMULATI.EVENT    TARGET OF ASSERTION: AASS120AC
90   SIMULATI.REQUEST       GROUP IN RECORD SIMULATI.EVENT
89   SIMULATI.EVENT         RECORD IN FILE SIMULATI           WRITE RECORD
                            END ITERATION                     FOR $I1
110  $YSGEN1                GROUP
111  $YSGEN2                GROUP
112  $YSGEN3                GROUP
57   ALLOC                  FILE                              CLOSE FILE
66   QUEUE                  FILE                              CLOSE FILE
32   SIMULATI               FILE                              CLOSE FILE
                            END
```

--- PL/I PROGRAM ---

```
R: PROCEDURE OPTIONS(MAIN);
DCL REQREL$ RECORD SEQL INPUT;
DCL $FSTREQREL$ BIT(1) INIT('1'B);
DCL EXIST_REQREL$ BIT(1) INIT('1'B);
DCL ENDFILE$REQREL$ BIT(1) INIT('0'B);
DCL REQREL_$ CHAR(19) VARYING INIT('');
DCL REQREL_INDX FIXED BIN;
DCL SIMULATI_HD1_$ CHAR(125) VARYING;
DCL SIMULATI_HD1_$_F CHAR(125);
DCL SIMULATI_HD1_$C BIT(1000) BASED(ADDR(SIMULATI_HD1_$_F));
DCL SIMULATI_HD1_INDX FIXED BIN;
DCL SIMULATI_HD2_$ CHAR(125) VARYING;
DCL SIMULATI_HD2_$_F CHAR(125);
DCL SIMULATI_HD2_$C BIT(1000) BASED(ADDR(SIMULATI_HD2_$_F));
DCL SIMULATI_HD2_INDX FIXED BIN;
DCL SIMULATI_HD3_$ CHAR(125) VARYING;
DCL SIMULATI_HD3_$_F CHAR(125);
DCL SIMULATI_HD3_$C BIT(1000) BASED(ADDR(SIMULATI_HD3_$_F));
DCL SIMULATI_HD3_INDX FIXED BIN;
DCL REQREL_MSGRM_$ CHAR(19) VARYING;
DCL REQREL_MSGRM_INDX FIXED BIN;
DCL ALLOC_MSGA_$ CHAR(20) VARYING;
DCL ALLOC_MSGA_$_F CHAR(20);
DCL ALLOC_MSGA_$C BIT(160) BASED(ADDR(ALLOC_MSGA_$_F));
DCL ALLOC_MSGA_INDX FIXED BIN;
DCL QUEUE_PROC_Q_$ CHAR(2574) VARYING;
DCL QUEUE_PROC_Q_$_F CHAR(2574);
DCL QUEUE_PROC_Q_$C BIT(20592) BASED(ADDR(QUEUE_PROC_Q_$_F));
DCL QUEUE_PROC_Q_INDX FIXED BIN;
DCL SIMULATI_EVENT_$ CHAR(1911) VARYING;
DCL SIMULATI_EVENT_$_F CHAR(1911);
DCL SIMULATI_EVENT_$C BIT(15288) BASED(ADDR(SIMULATI_EVENT_$_F));
DCL SIMULATI_EVENT_INDX FIXED BIN;
DCL $W_QUEUE$PROC_ID(2) FIXED BIN;
CALL $INITWIN($W_QUEUE$PROC_ID,2);
DCL $W_QUEUE$SAT(2) FIXED BIN;
CALL $INITWIN($W_QUEUE$SAT,2);
DCL $W_QUEUE$CLAIM(2) FIXED BIN;
CALL $INITWIN($W_QUEUE$CLAIM,2);
DCL ALLOCT RECORD SEQL OUTPUT;
DCL $FSTALLOCT BIT(1) INIT('1'B);
DCL QUEUET RECORD SEQL OUTPUT;
DCL $FSTQUEUET BIT(1) INIT('1'B);
DCL SIMULATIT RECORD SEQL OUTPUT;
DCL $FSTSIMULATIT BIT(1) INIT('1'B);
DCL $ERROR_BUF CHAR(270) VAR;
DCL ERRORF FILE RECORD OUTPUT;
DCL ERRORF_BIT BIT(1) STATIC INIT('1'B);
DCL $NOT_DONE(20) BIT(1);
DCL $ERROR BIT(1) INIT('0'B);
DCL $TMP_VAL FLOAT BIN;
DCL ($RD_LP$,$R_L) LABEL;
DECLARE
   1 ALLOC,
    2 MSGAS,
     3 MSGA,
      4 PROC_ID(99) CHAR(11),
      4 CLOCKA(99) PIC'999999999';
```

```
DECLARE
   1 QUEUE,
    2 PROC_Q,
     3 PROC,
      4 PROC_ID(2,99) PIC'9',
      4 IN_IX(99) PIC'999999999',
      4 OUT_IX(99) PIC'9',
      4 RES,
       5 CLAIM(2,99,5) PIC'9',
       5 SUM_REQ(99,5) PIC'9',
       5 SAT(2,99,5) BIT(1);
DECLARE
   1 REQREL,
    2 MSGRM,
     3 PROC_IDM PIC'9',
     3 RQ_OR_RL CHAR(3),
     3 CLOCKRM PIC'999999999',
     3 RESM PIC'9';
DECLARE
   1 SIMULATI,
    2 HD1,
     3 HDF1 CHAR(125),
    2 HD2,
     3 HDF2 CHAR(125),
    2 HD3,
     3 HDF3 CHAR(125),
    2 EVENT,
     3 REQUEST,
      4 PROC_IDM CHAR(4),
      4 FILLER1 CHAR(1),
      4 RQ_OR_RL CHAR(3),
      4 FILLER2 CHAR(1),
      4 RESM(5) PIC'9',
      4 FILLER3 CHAR(1),
      4 CLOCKRM PIC'B(12)9',
      4 FILLER4 CHAR(2),
     3 ALLOCATI,
      4 FILLER5(99) CHAR(1),
      4 PROC_IDA(99) CHAR(4),
      4 FILLER6(99) CHAR(1),
      4 CLOCKA(99) PIC'B(12)9';
DECLARE
   1 INTERIM,
    2 RES_LIMI,
     3 NUM_RES(5) PIC'9',
    2 $YSGEN1,
     3 END$REQREL_MSGRM(2) BIT(1) ,
    2 $YSGEN2,
     3 SIZE$QUEUE_PROC(2) FIXED BIN ,
    2 $YSGEN3,
     3 SIZE$ALLOC_MSGA FIXED BIN ;
DCL $I1 FIXED BIN;
DCL $I2 FIXED BIN;
DCL $I3 FIXED BIN;
DCL (TRUE,SELECTED) BIT(1) INIT('1'B);
DCL (FALSE,NOT_SELE,NOT_SELECTED) BIT(1) INIT('0'B);
DCL SUBSTR BUILTIN;
DCL DATE BUILTIN;
LEFTJ: PROC(K) RETURNS(CHAR(10) VAR); /* $START$ */
  /* CONVERT A NUMBER INTO A VARIABLE LENGTH STRING */
  DCL (k,j) FIXED BIN;
  DCL ST CHAR(20) VAR;
  dcl answer char(10) var;
  ST=K;
  J=VERIFY(ST,' 0');
  IF J=0 THEN ST='0';
         ELSE ST=SUBSTR(ST,J);
  answer=st;
  return(answer);
END LEFTJ;     /* $END$ */
```

```
$ROTATE: PROCEDURE(WIN_VEC,LEN): /* $START$ */
  DCL WIN_VEC(*) FIXED BIN;
  DCL (I,LEN,TEMP) FIXED BIN;
  TEMP=WIN_VEC(1);
  DO I = 2 TO LEN;
    WIN_VEC(I-1)=WIN_VEC(I);
  END;
  WIN_VEC(LEN)=TEMP;
END $ROTATE:   /* $END$ */
$INITWIN: PROCEDURE(WIN_VEC,LEN): /* $START$ */
  DCL WIN_VEC(*) FIXED BIN;
  DCL (I,LEN) FIXED BIN;
  DO I=1 TO LEN;
    WIN_VEC(I)=I;
  END;
END $INITWIN:   /* $END$ */
CHPTRLB: PROC(NAME) RETURNS (CHAR(32) VAR);  /* $START$ */
DCL NAME CHAR(*);
DCL RESULT CHAR(32) VAR;
DCL (I,LEN) FIXED BIN;
IF NAME=' ' THEN RETURN(' ');
LEN=LENGTH(NAME);
/*THIS PROCEDURE CHOPS OFF THE TRAILING BLANKS (CHTRLB) OF A FIXED
LENGTH 'NAME' AND RETURNS A VARIABLE LENGTH ONE*/
DO I=LEN TO 1 BY -1 WHILE(SUBSTR(NAME,I,1)=' ');
END;
RESULT=SUBSTR(NAME,1,I);
RETURN(RESULT);
END CHPTRLB:   /* $END$ */
ON ENDFILE(REQRELS) BEGIN;
goto $RD_LP;
 END;
ON UNDEFINEDFILE(ERRORF)  ERRORF_BIT='0'B;
DECLARE PLI$_CNVERR GLOBALREF VALUE FIXED BIN(31);
DECLARE RMS$_RLK GLOBALREF VALUE FIXED BIN(31);
ON ERROR BEGIN;
IF ONCODE()=RMS$_RLK THEN GOTO $RD_LP$;
IF ^(SERROR) THEN CALL RESIGNAL();
IF ONCODE()=PLI$_CNVERR THEN DO;
$ERROR='0'B;
IF ERRORF_BIT THEN WRITE FILE(ERRORF) FROM ($ERROR_BUF);
 END;
ELSE CALL RESIGNAL();
 END;
MAILBOX_NAME='REQRELS_MBX';
MAX_LENGTH=19;
STS$VALUE=SYS$CREMBX(PERMANENT,CHANNEL,MAX_LENGTH,,PROT_MASK,,MAILBOX_NAME);
LN_REQRELS='REQRELS';
EQV_NAME=' ';
STS$VALUE=SYS$TRNLOG(LN_REQRELS,L_LENGTH,EQV_NAME,,,);
if ^sts$success then put skip list('translation error');
OPEN FILE(REQRELS) INPUT TITLE(EQV_NAME);
$X$=SUBSTR(EQV_NAME,1,L_LENGTH);
STS$VALUE=SYS$ASSIGN($X$,CHANNEL,,,);
IF ^STS$SUCCESS THEN EXIST_REQRELS='0'B;
SIMULATI_HDF1='          REQUEST                                    ALLOCATION';
SIMULATI_HD1_INDX=1;
SUBSTR(SIMULATI_HD1_S_F,SIMULATI_HD1_INDX,125)=SIMULATI_HDF1 ;
SIMULATI_HD1_INDX=SIMULATI_HD1_INDX+125 ;
SIMULATI_HD1_S=SUBSTR(SIMULATI_HD1_S_F,1,SIMULATI_HD1_INDX-1);
WRITE FILE(SIMULATIT) FROM (SIMULATI_HD1_S);
DO $I1 =1 TO 5;
  INTERIM,NUM_RES($I1)=1;
END;
SIMULATI_HDF3='==== === ===== ============   ==== ============'!!' ==== =====
!!'========'!!' ==== ============';
SIMULATI_HDF2='P_ID R/L RESRC     TIME        P_ID     TIME     '!!' P_ID     T
!!'IME      '!!' P_ID     TIME';
```

```
SIMULATI_HD2_INDX=1;
SUBSTR(SIMULATI_HD2_S_F,SIMULATI_HD2_INDX,125)=SIMULATI.HDF2 ;
SIMULATI_HD2_INDX=SIMULATI_HD2_INDX+125 ;
SIMULATI_HD2_S=SUBSTR(SIMULATI_HD2_S_F,1,SIMULATI_HD2_INDX-1);
WRITE FILE(SIMULATIT) FROM (SIMULATI_HD2_S);
SIMULATI_HD3_INDX=1;
SUBSTR(SIMULATI_HD3_S_F,SIMULATI_HD3_INDX,125)=SIMULATI.HDF3 ;
SIMULATI_HD3_INDX=SIMULATI_HD3_INDX+125 ;
SIMULATI_HD3_S=SUBSTR(SIMULATI_HD3_S_F,1,SIMULATI_HD3_INDX-1);
WRITE FILE(SIMULATIT) FROM (SIMULATI_HD3_S);
$I1 =0;
$NOT_DONE(1)='1'B;
DO  WHILE($NOT_DONE(1));
  $I1 = $I1 +1;
  END$REQREL_MSGRM(2)=DATE)='840923';
  $RD_REQRELS;
  $R__=$RD_REQRELS;
    READ FILE(REQRELS) INTO (REQREL_MSGRM_S);
  REQREL_MSGRM_INDX=1;
  $ERROR_BUF=REQREL_MSGRM_S;
  REQREL_MSGRM_S=REQREL_MSGRM_S||copy(' ',19);
  $ERROR='1'B ;
  UNSPEC(REQREL.PROC_IDM)=UNSPEC(SUBSTR(REQREL_MSGRM_S,REQREL_MSGRM_INDX,1)) ;
  REQREL.PROC_IDM = REQREL.PROC_IDM ;
  IF $ERROR THEN $ERROR='0'B  ;
  REQREL_MSGRM_INDX=REQREL_MSGRM_INDX+1 ;
  SIMULATI.PROC_IDM=REQREL.PROC_IDM;
  REQREL.RQ_OR_RL=SUBSTR(REQREL_MSGRM_S,REQREL_MSGRM_INDX,3) ;
  REQREL_MSGRM_INDX=REQREL_MSGRM_INDX+3 ;
  SIMULATI.RQ_OR_RL=REQREL.RQ_OR_RL;
  IF $I1=1 THEN SIZE$QUEUE_PROC(2)=1;
   ELSE IF REQREL.RQ_OR_RL='REL' THEN SIZE$QUEUE_PROC(2)=SIZE$QUEUE_PROC(1)-1;
   ELSE SIZE$QUEUE_PROC(2)=SIZE$QUEUE_PROC(1)+1;
  DO $I2 =1 TO SIZE$QUEUE_PROC(2);
    IF REQREL.RQ_OR_RL='REQ' THEN QUEUE.IN_IX($I2)=$I2;
     ELSE IF REQREL.PROC_IDM^=QUEUE.PROC_ID($W_QUEUE$PROC_ID(1),$I2) THEN IF
    $I2=1 THEN QUEUE.IN_IX($I2)=1;
     ELSE QUEUE.IN_IX($I2)=QUEUE.IN_IX($I2-1)+1;
     ELSE IF $I2=1 THEN QUEUE.IN_IX($I2)=2;
     ELSE QUEUE.IN_IX($I2)=QUEUE.IN_IX($I2-1)+2;
    IF REQREL.RQ_OR_RL='REQ'&$I2=SIZE$QUEUE_PROC(2) THEN
    QUEUE.PROC_ID($W_QUEUE$PROC_ID(2),$I2)=REQREL.PROC_IDM;
     ELSE QUEUE.PROC_ID($W_QUEUE$PROC_ID(2),$I2)=
    QUEUE.PROC_ID($W_QUEUE$PROC_ID(1),QUEUE.IN_IX($I2));
  END;
  $ERROR='1'B ;
  UNSPEC(REQREL.CLOCKRM)=UNSPEC(SUBSTR(REQREL_MSGRM_S,REQREL_MSGRM_INDX,10)) ;
  REQREL.CLOCKRM = REQREL.CLOCKRM ;
  IF $ERROR THEN $ERROR='0'B  ;
  REQREL_MSGRM_INDX=REQREL_MSGRM_INDX+10 ;
  SIMULATI.CLOCKRM=REQREL.CLOCKRM;
  DO $I2 =1 TO 5;
    $ERROR='1'B ;
    UNSPEC(REQREL.RESM)=UNSPEC(SUBSTR(REQREL_MSGRM_S,REQREL_MSGRM_INDX,1)) ;
    REQREL.RESM = REQREL.RESM ;
    IF $ERROR THEN $ERROR='0'B  ;
    REQREL_MSGRM_INDX=REQREL_MSGRM_INDX+1 ;
    SIMULATI.RESM($I2)=REQREL.RESM;
    DO $I3 =1 TO SIZE$QUEUE_PROC(2);
      IF REQREL.RQ_OR_RL='REQ'&$I3=SIZE$QUEUE_PROC(2) THEN
      QUEUE.CLAIM($W_QUEUE$CLAIM(2),$I3,$I2)=REQREL.RESM;
       ELSE QUEUE.CLAIM($W_QUEUE$CLAIM(2),$I3,$I2)=QUEUE.CLAIM($W_QUEUE$CLAIM(1)
      ,QUEUE.IN_IX($I3),$I2);
      IF $I3=1 THEN QUEUE.SUM_REQ($I3,$I2)=QUEUE.CLAIM($W_QUEUE$CLAIM(2),$I3,
      $I2);
       ELSE QUEUE.SUM_REQ($I3,$I2)=QUEUE.CLAIM($W_QUEUE$CLAIM(2),$I3,$I2)+
      QUEUE.SUM_REQ($I3-1,$I2);
```

```
      IF $I2=1 THEN QUEUE.SAT($W_QUEUE$SAT(2),$I3,$I2)=QUEUE.SUM_REQ($I3,$I2)<=
      INTERIM.NUM_RES($I2);QUEUE.CLAIM($W_QUEUE$CLAIM(2),$I3,$I2)=0;
       ELSE QUEUE.SAT($W_QUEUE$SAT(2),$I3,$I2)=QUEUE.SAT($W_QUEUE$SAT(2),$I3,
      $I2-1)&QUEUE.SUM_REQ($I3,$I2)<=INTERIM.NUM_RES($I2);
      QUEUE.CLAIM($W_QUEUE$CLAIM(2),$I3,$I2)=0;
    END;
  END;
  DO $I2 =1 TO SIZE$QUEUE_PROC(2);
    IF REQREL.RQ_OR_RL='REL' THEN IF QUEUE.SAT($W_QUEUE$SAT(2),$I2,5)&^
    QUEUE.SAT($W_QUEUE$SAT(1),QUEUE.IN_IX($I2),5) THEN IF $I2=1 THEN
    QUEUE.OUT_IX($I2)=1;
     ELSE QUEUE.OUT_IX($I2)=QUEUE.OUT_IX($I2-1)+1;
     ELSE IF $I2=1 THEN QUEUE.OUT_IX($I2)=0;
     ELSE QUEUE.OUT_IX($I2)=QUEUE.OUT_IX($I2-1);
     ELSE IF $I2=SIZE$QUEUE_PROC(2)&QUEUE.SAT($W_QUEUE$SAT(2),$I2,5) THEN
    QUEUE.OUT_IX($I2)=1;
     ELSE QUEUE.OUT_IX($I2)=0;
    IF $I2=1&QUEUE.OUT_IX($I2)=1|$I2>1&QUEUE.OUT_IX($I2)>QUEUE.OUT_IX($I2-1)
    THEN ALLOC.CLOCKA(QUEUE.OUT_IX($I2))=REQREL.CLOCKRM;
     ELSE ;
    IF $I2=1&QUEUE.OUT_IX($I2)=1|$I2>1&QUEUE.OUT_IX($I2)>QUEUE.OUT_IX($I2-1)
    THEN ALLOC.PROC_ID(QUEUE.OUT_IX($I2))=CHPTRLB('ALLOC'||
    LEFTJ(QUEUE.PROC_ID($W_QUEUE$PROC_ID(2),$I2))||'S_MBX');
     ELSE ;
  END;
  IF SIZE$QUEUE_PROC(2)>0 THEN SIZE$ALLOC_MSGA=QUEUE.OUT_IX(SIZE$QUEUE_PROC(2));
   ELSE SIZE$ALLOC_MSGA=0;
  DO $I2 =1 TO SIZE$ALLOC_MSGA;
    SIMULATI.CLOCKA($I2)=ALLOC.CLOCKA($I2);
    SIMULATI.PROC_IDA($I2)=SUBSTR(ALLOC.PROC_ID($I2),6,1);
    ALLOC_MSGA_INDX=1;
    SUBSTR(ALLOC_MSGA_S_F,ALLOC_MSGA_INDX,11)=ALLOC.PROC_ID($I2) ;
    ALLOC_MSGA_INDX=ALLOC_MSGA_INDX+11 ;
    SUBSTR(ALLOC_MSGA_SC,ALLOC_MSGA_INDX*8-7,9*8)=UNSPEC(ALLOC.CLOCKA($I2)) ;
    ALLOC_MSGA_INDX=ALLOC_MSGA_INDX+9 ;
    ALLOC_MSGA_S=SUBSTR(ALLOC_MSGA_S_F,1,ALLOC_MSGA_INDX-1);
    STS$VALUE=SYS$ASSIGN(CHPTRLB(ALLOC.PROC_ID($I2)),CHANNEL,,,);
    IF ^STS$SUCCESS THEN DO;
    PUT SKIP LIST('**WARNING: NO CHANNEL ASSIGNED TO ALLOCT');
    PUT SKIP LIST('**          MESSAGE DEPOSITED INTO ALLOCT.DAT');
    WRITE FILE(ALLOCT) FROM (ALLOC_MSGA_S);
  END;
   ELSE DO;
  STS$VALUE=SYS$QIO(1,CHANNEL,IO$_WRITEVBLK+IO$M_NOW,IO_STATUS,,,
  ADDR(ALLOC_MSGA_S_F),LENGTH(ALLOC_MSGA_S),,,,,);
  IF ^STS$SUCCESS THEN PUT SKIP LIST('WRITE QIO UNSUCCESSFUL');
  STS$VALUE=SYS$WAITFR(1);
  STS$VALUE=SYS$DASSGN(CHANNEL);
END;
SIMULATI.FILLER5($I2)=' ';
SIMULATI.FILLER6($I2)=' ';
END;
QUEUE_PROC_Q_INDX=1;
DO $I2 =1 TO SIZE$QUEUE_PROC(2);
  SUBSTR(QUEUE_PROC_Q_SC,QUEUE_PROC_Q_INDX*8-7,1*8)=
  UNSPEC(QUEUE.PROC_ID($W_QUEUE$PROC_ID(2),$I2)) ;
  QUEUE_PROC_Q_INDX=QUEUE_PROC_Q_INDX+1 ;
  SUBSTR(QUEUE_PROC_Q_SC,QUEUE_PROC_Q_INDX*8-7,9*8)=UNSPEC(QUEUE.IN_IX($I2)) ;
  QUEUE_PROC_Q_INDX=QUEUE_PROC_Q_INDX+9 ;
```

```
  SUBSTR(QUEUE_PROC_Q_SC,QUEUE_PROC_Q_INDX*8-7,1*8)=UNSPEC(QUEUE.OUT_IT($I2)) :
  QUEUE_PROC_Q_INDX=QUEUE_PROC_Q_INDX+1 :
  DO $I3 =1 TO 5:
    SUBSTR(QUEUE_PROC_Q_SC,QUEUE_PROC_Q_INDX*8-7,1*8)=
    UNSPEC(QUEUE.CLAIM($W_QUEUE$CLAIM(2),$I2,$I3)) :
    QUEUE_PROC_Q_INDX=QUEUE_PROC_Q_INDX+1 :
    SUBSTR(QUEUE_PROC_Q_SC,QUEUE_PROC_Q_INDX*8-7,1*8)=UNSPEC(QUEUE.SUM_REQ($I2,
    $I3)) :
    QUEUE_PROC_Q_INDX=QUEUE_PROC_Q_INDX+1 :
    SUBSTR(QUEUE_PROC_Q_SC,QUEUE_PROC_Q_INDX*8-7,1*8)=
    UNSPEC(QUEUE.SAT($W_QUEUE$SAT(2),$I2,$I3)) :
    QUEUE_PROC_Q_INDX=QUEUE_PROC_Q_INDX+1 :
  END:
END:
QUEUE_PROC_Q_S=SUBSTR(QUEUE_PROC_Q_S_F,1,QUEUE_PROC_Q_INDX-1):
WRITE FILE(QUEUET) FROM (QUEUE_PROC_Q_S):
SIMULATI.FILLER1=' ':
SIMULATI.FILLER2=' ':
SIMULATI.FILLER3=' ':
SIMULATI.FILLER4=' ':
SIMULATI_EVENT_INDX=1:
SUBSTR(SIMULATI_EVENT_S_F,SIMULATI_EVENT_INDX,4)=SIMULATI.PROC_IDM :
SIMULATI_EVENT_INDX=SIMULATI_EVENT_INDX+4 :
SUBSTR(SIMULATI_EVENT_S_F,SIMULATI_EVENT_INDX,1)=SIMULATI.FILLER1 :
SIMULATI_EVENT_INDX=SIMULATI_EVENT_INDX+1 :
SUBSTR(SIMULATI_EVENT_S_F,SIMULATI_EVENT_INDX,3)=SIMULATI.RQ_OR_RL :
SIMULATI_EVENT_INDX=SIMULATI_EVENT_INDX+3 :
SUBSTR(SIMULATI_EVENT_S_F,SIMULATI_EVENT_INDX,1)=SIMULATI.FILLER2 :
SIMULATI_EVENT_INDX=SIMULATI_EVENT_INDX+1 :
DO $I2 =1 TO 5:
  SUBSTR(SIMULATI_EVENT_SC,SIMULATI_EVENT_INDX*8-7,1*8)=
  UNSPEC(SIMULATI.RESM($I2)) :
  SIMULATI_EVENT_INDX=SIMULATI_EVENT_INDX+1 :
END:
SUBSTR(SIMULATI_EVENT_S_F,SIMULATI_EVENT_INDX,1)=SIMULATI.FILLER3 :
SIMULATI_EVENT_INDX=SIMULATI_EVENT_INDX+1 :
SUBSTR(SIMULATI_EVENT_SC,SIMULATI_EVENT_INDX*8-7,13*8)=UNSPEC(SIMULATI.CLOCKRM)
:
SIMULATI_EVENT_INDX=SIMULATI_EVENT_INDX+13 :
SUBSTR(SIMULATI_EVENT_S_F,SIMULATI_EVENT_INDX,2)=SIMULATI.FILLER4 :
SIMULATI_EVENT_INDX=SIMULATI_EVENT_INDX+2 :
DO $I2 =1 TO SIZE$ALLOC_MSGA:
  SUBSTR(SIMULATI_EVENT_S_F,SIMULATI_EVENT_INDX,1)=SIMULATI.FILLER5($I2) :
  SIMULATI_EVENT_INDX=SIMULATI_EVENT_INDX+1 :
  SUBSTR(SIMULATI_EVENT_S_F,SIMULATI_EVENT_INDX,4)=SIMULATI.PROC_IDA($I2) :
  SIMULATI_EVENT_INDX=SIMULATI_EVENT_INDX+4 :
  SUBSTR(SIMULATI_EVENT_S_F,SIMULATI_EVENT_INDX,1)=SIMULATI.FILLER6($I2) :
  SIMULATI_EVENT_INDX=SIMULATI_EVENT_INDX+1 :
  SUBSTR(SIMULATI_EVENT_SC,SIMULATI_EVENT_INDX*8-7,13*8)=
  UNSPEC(SIMULATI.CLOCKA($I2)) :
  SIMULATI_EVENT_INDX=SIMULATI_EVENT_INDX+13 :
END:
SIMULATI_EVENT_S=SUBSTR(SIMULATI_EVENT_S_F,1,SIMULATI_EVENT_INDX-1):
WRITE FILE(SIMULATIT) FROM (SIMULATI_EVENT_S):
IF END$REQREL_MSGRM(2) THEN $NOT_DONE(1)='0'B:
SIZE$QUEUE_PROC(1) = SIZE$QUEUE_PROC(2):
END$REQREL_MSGRM(1) = END$REQREL_MSGRM(2):
CALL $ROTATE('$W_QUEUE$PROC_ID',2):
CALL $ROTATE('$W_QUEUE$SAT',2):
CALL $ROTATE('$W_QUEUE$CLAIM',2):
END:
CLOSE FILE(ALLOCT):
CLOSE FILE(QUEUET):
CLOSE FILE(SIMULATIT):
RETURN:
END P:
```

ALL CONFIGURATION DOCUMENTATION

```
*************************************************************************
*                        LISTING OF SPECIFICATION                       *
*************************************************************************

    1     CONFIGURATION: FR:

    2     M: P1,P2,P3,P4,P5 -> F: REQREL (ORG:MAIL)
    3        -> M: MONITOR -> F: ALLOCT (ORG:POST)
    4        -> F: ALLOC1S (ORG:MAIL), ALLOC2S (ORG:MAIL), ALLOC3S (ORG:MAIL),
    5              ALLOC4S (ORG:MAIL), ALLOC5S (ORG:MAIL):

    6     F: ALLOC1S  -> M: P1:
    7     F: ALLOC2S  -> M: P2:
    8     F: ALLOC3S  -> M: P3:
    9     F: ALLOC4S  -> M: P4:
   10     F: ALLOC5S  -> M: P5:
```

```
*********************************
*     CONFIGURATION REPORT      *
*********************************

                    |          !111
      NAME          |123456789012345
====================|===============
    MODULE(S)   *   |
--------------------|
 1 MONITOR          |            V
 2 P1               |             V
 3 P2               |             V
 4 P3               |             V
 5 P4               |             V
 6 P5               |             V
====================|===============
    FILE(S)     *   |
--------------------|
 7 ALLOC1S          | V
 8 ALLOC2S          |  V
 9 ALLOC3S          |   V
10 ALLOC4S          |    V
11 ALLOC5S          |     V
12 ALLOCT           |      VVVVV
13 REQREL          :V
```

```
*********************************************
*              MSCC REPORT                  *
*********************************************

  MSCC#      NODE NAME       TYPE
=======    ===========     ====
      1    MONITOR         MDL
           ALLOCT          FILE
           ALLOC1S         FILE
           P1              MDL
           REQREL          FILE
           ALLOC2S         FILE
           P2              MDL
           ALLOC3S         FILE
           P3              MDL
           ALLOC4S         FILE
           P4              MDL
           ALLOC5S         FILE
           P5              MDL
```

```
*************************************************************************
*                          MDL/FILE REF REPORT                          *
*************************************************************************

-- MODULE X-REF --

MDL: MONITOR
     P_NAME: MONITOR

   SYNONYMS          SOURCE              TARGET          SYNC_AFTER  PAR#: 1
  ==========  ================== ================== ==========  ==========
              REQREL     /VIRTUAL ALLOCT    /VIRTUAL              P5
                                                                  P4
                                                                  P3
                                                                  P2
                                                                  P1
                                                                  MONITOR
```

```
MDL: P1
     P_NAME: P1

    SYNONYMS          SOURCE              TARGET          SYNC_AFTER  PAR#: 1
    ==========  ================== ==================  ==========  ==========
                ALLOC1S   /VIRTUAL  REQREL    /VIRTUAL                P5
                                                                    P4
                                                                    P3
                                                                    P2
                                                                    P1
                                                                    MONITOR
MDL: P2
     P_NAME: P2

    SYNONYMS          SOURCE              TARGET          SYNC_AFTER  PAR#: 1
    ==========  ================== ==================  ==========  ==========
                ALLOC2S   /VIRTUAL  REQREL    /VIRTUAL                P5
                                                                    P4
                                                                    P3
                                                                    P2
                                                                    P1
                                                                    MONITOR
MDL: P3
     P_NAME: P3

    SYNONYMS          SOURCE              TARGET          SYNC_AFTER  PAR#: 1
    ==========  ================== ==================  ==========  ==========
                ALLOC3S   /VIRTUAL  REQREL    /VIRTUAL                P5
                                                                    P4
                                                                    P3
                                                                    P2
                                                                    P1
                                                                    MONITOR
MDL: P4
     P_NAME: P4

    SYNONYMS          SOURCE              TARGET          SYNC_AFTER  PAR#: 1
    ==========  ================== ==================  ==========  ==========
                ALLOC4S   /VIRTUAL  REQREL    /VIRTUAL                P5
                                                                    P4
                                                                    P3
                                                                    P2
                                                                    P1
                                                                    MONITOR
MDL: P5
     P_NAME: P5

    SYNONYMS          SOURCE              TARGET          SYNC_AFTER  PAR#: 1
    ==========  ================== ==================  ==========  ==========
                ALLOC5S   /VIRTUAL  REQREL    /VIRTUAL                P5
                                                                    P4
                                                                    P3
                                                                    P2
                                                                    P1
                                                                    MONITOR
-- FILE X-REF --

FILE: ALLOC1S
     P_NAME: ALLOC1S

              (COMPATIBILITY REQUIRED)
    SYNONYMS   PRODUCER   CONSUMBER   ORGANIZATION  REC SIZE   PAR#: 1
    ==========  ==========  ==========  ============  ========  ==========
                ALLOCT      P1          MAIL/VIRTUAL  300       P5
                                                                P4
                                                                P3
                                                                P2
                                                                P1
                                                                MONITOR
```

FILE: ALLOC2S
P_NAME: ALLOC2S

SYNONYMS	(COMPATIBILITY REQUIRED) PRODUCER	CONSUMBER	ORGANIZATION	REC SIZE	PAR#: 1
	ALLOCT	P2	MAIL/VIRTUAL	300	P5 P4 P3 P2 P1 MONITOR

FILE: ALLOC3S
P_NAME: ALLOC3S

SYNONYMS	(COMPATIBILITY REQUIRED) PRODUCER	CONSUMBER	ORGANIZATION	REC SIZE	PAR#: 1
	ALLOCT	P3	MAIL/VIRTUAL	300	P5 P4 P3 P2 P1 MONITOR

FILE: ALLOC4S
P_NAME: ALLOC4S

SYNONYMS	(COMPATIBILITY REQUIRED) PRODUCER	CONSUMBER	ORGANIZATION	REC SIZE	PAR#: 1
	ALLOCT	P4	MAIL/VIRTUAL	300	P5 P4 P3 P2 P1 MONITOR

FILE: ALLOC5S
P_NAME: ALLOC5S

SYNONYMS	(COMPATIBILITY REQUIRED) PRODUCER	CONSUMBER	ORGANIZATION	REC SIZE	PAR#: 1
	ALLOCT	P5	MAIL/VIRTUAL	300	P5 P4 P3 P2 P1 MONITOR

FILE: ALLOCT
P_NAME: ALLOCT

SYNONYMS	(COMPATIBILITY REQUIRED) PRODUCER	CONSUMBER	ORGANIZATION	REC SIZE	PAR#: 1
	MONITOR	ALLOC1S ALLOC2S ALLOC3S ALLOC4S ALLOC5S	POST/VIRTUAL	300	P5 P4 P3 P2 P1 MONITOR

FILE: REQREL
P_NAME: REQREL

SYNONYMS	(COMPATIBILITY REQUIRED) PRODUCER	CONSUMBER	ORGANIZATION	REC SIZE	PAR#: 1
	P1	MONITOR	MAIL/VIRTUAL	300	P5
	P2				P4
	P3				P3
	P4				P2
	P5				P1
					MONITOR

****
* CROSS REFERENCE REPORT *
****

NAME	M/F	TYPE	REP_NAME	PHSICAL NAME INFORMATION	WHERE IT IS DEFINED
MONITOR	M	MDL	MONITOR	MONITOR	3
P1	M	MDL	P1	P1	6 2
P2	M	MDL	P2	P2	7 3
P3	M	MDL	P3	P3	8 3
P4	M	MDL	P4	P4	9 3
P5	M	MDL	P5	P5	10 2
ALLOC1S	F	MAIL	ALLOC1S	ALLOC1S	6 4
ALLOC2S	F	MAIL	ALLOC2S	ALLOC2S	7 4
ALLOC3S	F	MAIL	ALLOC3S	ALLOC3S	8 4
ALLOC4S	F	MAIL	ALLOC4S	ALLOC4S	9 5
ALLOC5S	F	MAIL	ALLOC5S	ALLOC5S	10 5
ALLOCT	F	POST	ALLOCT	ALLOCT	4
REQREL	F	MAIL	REQREL	REQREL	3

****
* PAR. COMPONENT REPORT *
****

* 1. The PAR. COMPs: *

PAR. COMP#	NODE NAME	TYPE	ACTION	PHYSICAL NAME
1	MONITOR	MDL	SUBMIT	MONITOR
	ALLOCT	FILE	------	ALLOCT
	REQREL	FILE	------	REQREL
	ALLOC1S	FILE	------	ALLOC1S
	ALLOC2S	FILE	------	ALLOC2S
	ALLOC3S	FILE	------	ALLOC3S
	ALLOC4S	FILE	------	ALLOC4S
	ALLOC5S	FILE	------	ALLOC5S
	P1	MDL	SUBMIT	P1
	P2	MDL	SUBMIT	P2
	P3	MDL	SUBMIT	P3
	P4	MDL	SUBMIT	P4
	P5	MDL	SUBMIT	P5

* 2. Parallel componet precedence matrix *

PAR. COMP#	1
1	

```
*****************************************
*            JCL PROGRAM REPORT         *
*****************************************
```

1. The main JCL program (RR.COM)

```
$PLI RR
$LINK RR
$RUN RR
$SUBMIT/NAME=MONITOR MONITOR
$SUBMIT/NAME=P1 P1
$SUBMIT/NAME=P2 P2
$SUBMIT/NAME=P3 P3
$SUBMIT/NAME=P4 P4
$SUBMIT/NAME=P5 P5
$! END OF ONE PAR COMP
```

2. The JCL programs for each module

* JCL Program for MONITOR (MONITOR.COM)

```
$DEFINE REQREL REQREL_MBX
$RUN MONITOR
$DEASS  REQREL
```

* JCL Program for P1 (P1.COM)

```
$DEFINE ALLOC1S ALLOC1S_MBX
$DEFINE REQREL REQREL_MBX
$RUN P1
$DEASS  ALLOC1S
$DEASS REQREL
```

* JCL Program for P2 (P2.COM)

```
$DEFINE ALLOC2S ALLOC2S_MBX
$DEFINE REQREL REQREL_MBX
$RUN P2
$DEASS  ALLOC2S
$DEASS REQREL
```

* JCL Program for P3 (P3.COM)

```
$DEFINE ALLOC3S ALLOC3S_MBX
$DEFINE REQREL REQREL_MBX
$RUN P3
$DEASS  ALLOC3S
$DEASS REQREL
```

* JCL Program for P4 (P4.COM)

```
$DEFINE ALLOC4S ALLOC4S_MBX
$DEFINE REQREL REQREL_MBX
$RUN P4
$DEASS  ALLOC4S
$DEASS REQREL
```

* JCL Program for P5 (P5.COM)

```
$DEFINE ALLOC5S ALLOC5S_MBX
$DEFINE REQREL REQREL_MBX
$RUN P5
$DEASS  ALLOC5S
$DEASS REQREL
```

3. The mailbox creation PL/I program (RR.PLI)

```
CREMBX:PROC OPTIONS(MAIN) RETURNS(FIXED BIN(31));
%INCLUDE SYS$CREMBX;
%INCLUDE SYS$ASSIGN;
%INCLUDE $STSDEF;
DCL MBX_NAME CHAR(15) VAR;
DCL PERMANENT FIXED BIN(31) STATIC INIT(1),
    CHANNEL FIXED BIN(15),
    MAX_LENGTH FIXED BIN(31) STATIC,
    PROT_MASK BIT(16) ALIGNED STATIC INIT('FF00'B4),
    MAILBOX_NAME STATIC CHAR(15) VAR;
MAILBOX_NAME='ALLOC1S_MBX';
MAX_LENGTH=300;
```

```
STS$VALUE=SYS$CREMBX(PERMANENT,CHANNEL,
        MAX_LENGTH,,PROT_MASK,,MAILBOX_NAME);
MAILBOX_NAME='ALLOC2S_MBX';
MAX_LENGTH=300;
STS$VALUE=SYS$CREMBX(PERMANENT,CHANNEL,
        MAX_LENGTH,,PROT_MASK,,MAILBOX_NAME);
MAILBOX_NAME='ALLOC3S_MBX';
MAX_LENGTH=300;
STS$VALUE=SYS$CREMBX(PERMANENT,CHANNEL,
        MAX_LENGTH,,PROT_MASK,,MAILBOX_NAME);
MAILBOX_NAME='ALLOC4S_MBX';
MAX_LENGTH=300;
STS$VALUE=SYS$CREMBX(PERMANENT,CHANNEL,
        MAX_LENGTH,,PROT_MASK,,MAILBOX_NAME);
MAILBOX_NAME='ALLOC5S_MBX';
MAX_LENGTH=300;
STS$VALUE=SYS$CREMBX(PERMANENT,CHANNEL,
        MAX_LENGTH,,PROT_MASK,,MAILBOX_NAME);
MAILBOX_NAME='REQREL_MBX';
MAX_LENGTH=300;
STS$VALUE=SYS$CREMBX(PERMANENT,CHANNEL,
        MAX_LENGTH,,PROT_MASK,,MAILBOX_NAME);
RETURN(STS$VALUE);
END CREMBX;
```

4. The mailbox deletion PL/I program (RRD.PLI)

```
DELMBX:PROC OPTIONS(MAIN) RETURNS(FIXED BIN(31));
%INCLUDE SYS$DELMBX;
%INCLUDE SYS$ASSIGN;
%INCLUDE $STSDEF;
DCL MBX_NAME CHAR(15) VAR;
DCL PERMANENT FIXED BIN(31) STATIC INIT(1),
    CHANNEL FIXED BIN(15),
    MAX_LENGTH FIXED BIN(31) STATIC,
    PROT_MASK BIT(16) ALIGNED STATIC INIT('FF00'B4),
    MAILBOX_NAME STATIC CHAR(15) VAR;
MAILBOX_NAME='ALLOC1S_MBX';
MAX_LENGTH=300;
STS$VALUE=SYS$ASSIGN(MAILBOX_NAME,CHANNEL,,,);
IF ^STS$SUCCESS THEN PUT SKIP LIST('ERROR IN DELMBX');
STS$VALUE=SYS$DELMBX(CHANNEL);
MAILBOX_NAME='ALLOC2S_MBX';
MAX_LENGTH=300;
STS$VALUE=SYS$ASSIGN(MAILBOX_NAME,CHANNEL,,,);
IF ^STS$SUCCESS THEN PUT SKIP LIST('ERROR IN DELMBX');
STS$VALUE=SYS$DELMBX(CHANNEL);
MAILBOX_NAME='ALLOC3S_MBX';
MAX_LENGTH=300;
STS$VALUE=SYS$ASSIGN(MAILBOX_NAME,CHANNEL,,,);
IF ^STS$SUCCESS THEN PUT SKIP LIST('ERROR IN DELMBX');
STS$VALUE=SYS$DELMBX(CHANNEL);
MAILBOX_NAME='ALLOC4S_MBX';
MAX_LENGTH=300;
STS$VALUE=SYS$ASSIGN(MAILBOX_NAME,CHANNEL,,,);
IF ^STS$SUCCESS THEN PUT SKIP LIST('ERROR IN DELMBX');
STS$VALUE=SYS$DELMBX(CHANNEL);
MAILBOX_NAME='ALLOC5S_MBX';
MAX_LENGTH=300;
STS$VALUE=SYS$ASSIGN(MAILBOX_NAME,CHANNEL,,,);
IF ^STS$SUCCESS THEN PUT SKIP LIST('ERROR IN DELMBX');
STS$VALUE=SYS$DELMBX(CHANNEL);
MAILBOX_NAME='REQREL_MBX';
MAX_LENGTH=300;
STS$VALUE=SYS$ASSIGN(MAILBOX_NAME,CHANNEL,,,);
IF ^STS$SUCCESS THEN PUT SKIP LIST('ERROR IN DELMBX');
STS$VALUE=SYS$DELMBX(CHANNEL);
RETURN(STS$VALUE);
END DELMBX;
```

A1.4 SIMULATION REPORT

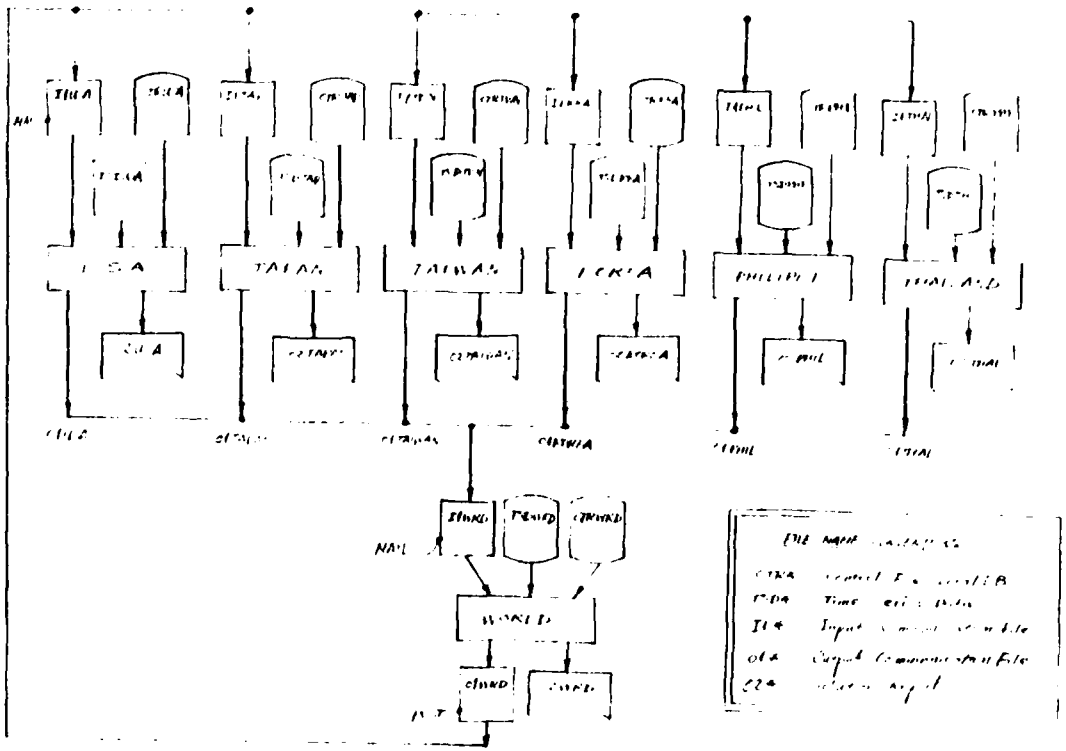
P_ID	R/L	REQUEST RESRC	TIME	P_ID	ALLOCATION TIME	P_ID	TIME	P_ID	TIME
2	REQ	01100	000015461434	2	000015461434				
1	REQ	11000	000015462017						
3	REQ	00110	000015462374						
5	REQ	10001	000015463088						
4	REQ	00011	000015463198						
2	REL	01100	000015463548	1	000015463548	3	000015463548		
2	REQ	01100	000015463549						
1	REL	11000	000015463611	5	000015463611				
1	REQ	11000	000015463612						
3	REL	00110	000015463615	2	000015463615				
3	REQ	00110	000015463617						
5	REL	10001	000015463639	4	000015463639				
5	REQ	10001	000015463640						
2	REL	01100	000015463662	1	000015463662				
2	REQ	01100	000015463664						
4	REL	00011	000015463686	3	000015463686				
4	REQ	00011	000015463687						
1	REL	11000	000015463709	5	000015463709				
1	REQ	11000	000015463711						
3	REL	00110	000015463733	2	000015463733				
3	REQ	00110	000015463735						
5	REL	10001	000015463757	4	000015463757				
5	REQ	10001	000015463758						
2	REL	01100	000015463782	1	000015463782				
2	REQ	01100	000015463784						
4	REL	00011	000015463805	3	000015463805				
4	REQ	00011	000015463807						
1	REL	11000	000015463829	5	000015463829				
1	REQ	11000	000015463831						
3	REL	00110	000015463858	2	000015463858				
3	REQ	00110	000015463860						
5	REL	10001	000015463884	4	000015463884				
5	REQ	10001	000015463885						
2	REL	01100	000015463909	1	000015463909				
2	REQ	01100	000015463910						
4	REL	00011	000015463935	3	000015463935				
4	REQ	00011	000015463936						
1	REL	11000	000015463958	5	000015463958				
1	REQ	11000	000015463959						
3	REL	00110	000015463983	2	000015463983				
3	REQ	00110	000015463984						
5	REL	10001	000015464006	4	000015464006				
5	REQ	10001	000015464008						
2	REL	01100	000015464030	1	000015464030				
2	REQ	01100	000015464032						
4	REL	00011	000015464058	3	000015464058				
4	REQ	00011	000015464064						
1	REL	11000	000015464103	5	000015464103				
1	REQ	11000	000015464104						
3	REL	00110	000015464165	2	000015464165				
3	REQ	00110	000015464166						
5	REL	10001	000015464207	4	000015464207				
5	REQ	10001	000015464209						
2	REL	01100	000015464239	1	000015464239				
4	REL	00011	000015464311	3	000015464311				
4	REQ	00011	000015464312						
1	REL	11000	000015464347	5	000015464347				
3	REL	00110	000015464385						
5	REL	10001	000015464439	4	000015464439				
4	REL	00011	000015464457						
$									

A2. THE COOPERATIVE COMPUTATION EXAMPLE

The example presented in Part I is an extremely simplified one. The real-life example of the cooperative computation is presented as the follows.

A2.1 SYSTEM CONFIGURATION

```
CONFIGURATION: PBEM;  /* PACIFIC BASIN ECONOMETRIC MODEL */
F: TSDUSAS,CTRUSAS,I1USAS (ORG: MAIL)
   -> M: USA
   -> F: O2USAT, O1USAT (ORG: MAIL);
F: TSDJAPANS,CTRJAPANS,I1JAPANS (ORG: MAIL)
   -> M: JAPAN
   -> F: O2JAPANT, O1JAPANT (ORG: MAIL);
F: TSDTWNS,CTRTWNS,I1TAIWANS (ORG: MAIL)
   -> M: TAIWAN
   -> F: O2TAIWANT, O1TAIWANT (ORG: MAIL);
F: TSDPHILS, CTRPHILS,I1PHILS (ORG: MAIL)
   -> M: PHILIPPI
   -> F: O2PHILT, O1PHILT (ORG: MAIL);
F: TSDTHAIS,CTRTHIS,I1THAIS (ORG: MAIL)
   -> M: THAILAND
   -> F: O2THAIT, O1THAIT (ORG: MAIL);
F: TSDKOREAS,CTRKRA,I1KOREAS (ORG: MAIL)
   -> M: KOREA
   -> F: O2KOREAT, O1KOREAT (ORG: MAIL);
F: TSDWRDS,CTRWRDS,
   USAS (ORG:MAIL),
   JAPS (ORG:MAIL),
   KRAS (ORG:MAIL),
   TWNS (ORG:MAIL),
   PHIS (ORG:MAIL),
   TAIS (ORG:MAIL)
   -> M: WORLD
   -> F: O2WORLDT, O1WORLDT (ORG: POST);
F: O1WORLDT
   -> F: I1USAS,I1JAPANS,I1TAIWANS,I1PHILS,I1KOREAS,I1THAIS;
S: O1USAT,USAS;
S: O1JAPANT,JAPS;
S: O1TAIWANT,TWNS;
S: O1KOREAT,KRAS;
S: O1PHILT,PHIS;
S: O1THAIT,TAIS;
```



THE GRAPH OF PBEM SYSTEM CONFIGURATION

A2.2 THE WORLD MODULE

```
MODULE: WORLD;
SOURCE: USA,JAP,KRA,TWN,PHI,TAI,TSDWRD, CTRWRD;
TARGET: 01WORLD,02WORLD;

1 USA FILE ORG MAIL,
  2 USARC(1:10) RECORD,
    4 PROC_ID FLD (CHAR 10),   /* MAIL ADDRESS */
    4 (M$2,PX$2,X$2,USAPY,USAY,USAR)
        ARE FLD (DEC FLOAT(15));

END.USARC(T)=T=SIM_PD;

1 JAP FILE ORG MAIL,
  2 JAPRC(1:10) RECORD,
    4 PROC_ID FLD (CHAR 10),   /* MAIL ADDRESS */
    4 (M$1,PX$1,X$1,JAPPY,JAPY,JAPR)
        ARE FLD (DEC FLOAT(15));

END.JAPRC(T)=T=SIM_PD;

1 TWN FILE ORG MAIL,
  2 TWNRC(1:10) RECORD,
    4 PROC_ID FLD (CHAR 10),   /* MAIL ADDRESS */
    4 (M$3,PX$3,X$3,TWNPY,TWNY,TWNR)
        ARE FLD (DEC FLOAT(15));

END.TWNRC(T)=T=SIM_PD;

1 KRA FILE ORG MAIL,
  2 KRARC(1:10) RECORD,
    4 PROC_ID FLD (CHAR 10),   /* MAIL ADDRESS */
    4 (M$4,PX$4,X$4,KRAPY,KRAY,KRAR)
        ARE FLD (DEC FLOAT(15));

END.KRARC(T)=T=SIM_PD;

1 PHI FILE ORG MAIL,
  2 PHIRC(1:10) RECORD,
    4 PROC_ID FLD (CHAR 10),   /* MAIL ADDRESS */
    4 (M$5,PX$5,X$5,PHIPY,PHIY,PHIR)
        ARE FLD (DEC FLOAT(15));

END.PHIRC(T)=T=SIM_PD;

1 TAI FILE ORG MAIL,
  2 TAIRC(1:10) RECORD,
    4 PROC_ID FLD (CHAR 10),   /* MAIL ADDRESS */
    4 (M$6,PX$6,X$6,THIPY,THIY,THIR)
        ARE FLD (DEC FLOAT(15));

END.TAIRC(T)=T=SIM_PD;

END.I2R(T)=T=SIM_PD;

1 CTRWRD FILE, /* THE CONTROL FILE */
  2 CR RECORD,
    3 START_YR FLD (PIC '9999'), /* STARTING YEAR OF SIMULATION */
    3 LAG      FLD (PIC '9'),    /* LAG FROM THE STARTING YEAR */
    3 SIM_PD   FLD (PIC '99'),   /* SIMULATION PERIOD.         */
  2 I21R RECORD,
    3 HDH FLD (CHAR 319),
  2 I2R(1:10) RECORD,            /* LOCAL HISTORICAL DATABASE  */
    3 HYR FLD (CHAR 4),
    3 (X$;1,X$22,X$33,X$44,X$55,X$66,ALPHA,BETA71,BETA72,BETA73,BETA74,BETA75,
```

```
      BETA76,BETA77,PX$7)
      ARE FLD (PIC'BB(10)-9V.(7)9');

1 TSDWRD FILE, /* TIME SERIES DATA FILE */
  2 TSR(1:100) GRP,
    3 HDR RECORD,
      4 HDD FLD (CHAR 21),
    3 TR(10) RECORD,
      4 TS_DATA FLD (PIC'BB(10)-9V.(7)9');

1 01WORLD FILE KEY IS ADR ORG POST,
  /* A UNIFORM FILE FOR SENDING BACK TO MODULES */
  2 01G(1:10) GRP,
    3 01R(6) RECORD,
      4 ADR FLD (CHAR 14), /* MAILING ADDRESSES */
      4 (CWT$,CPW$,PM$Z)
        ARE FLD (DEC FLOAT(15));

END.01G(T)=T=SIM_PD;

! INTX FILE, /* INTEGRATED FILE BEOFRE DISTRIBUTION */
  2 IN2R(1:10) RECORD,
    3 (WT$,PW$,PM$1,PM$2,PM$3,PM$4,PM$5,PM$6)
      ARE FLD (DEC FLOAT(15));

END.IN2R(T)=T=SIM_PD;

(T,I,J) SUBSCRIPT;

1 02WORLD FILE,  /* LOCAL RESULTS */
  3 HD RECORD,
    4 FHD FLD (CHAR 124),
  3 BLK0 RECORD,
    4 F0 FLD (CHAR 80),
  3 GGRP GRP,
    4 END_HD RECORD,
      5 END_HDN FLD (CHAR 124),
    4 END_FR RECORD,
      5 END_FF FLD (CHAR 80),
    4 NAMES RECORD,
      5 NM1 FLD (CHAR 124),
    4 BLK1 RECORD,
      5 F1 FLD (CHAR 80),
    4 VALUES1(1:10) RECORD,
      5 YEAR1 FLD (PIC '9999'),
      5 (X$31, X$51, X$61, X$12, X$32, X$52, X$13)
       FLD (PIC'BB(7)-9V.(6)9'),
    4 BLK11 RECORD,
      5 F11 FLD (CHAR 80),
    4 NAMES2 RECORD,
      5 NM2 FLD (CHAR 124),
    4 BLK2 RECORD,
      5 F2 FLD (CHAR 80),
    4 VALUES2(1:10) RECORD,
      5 YEAR2 FLD (PIC '9999'),
      5 (X$53, X$63, X$34, X$54, X$64, X$35, X$65)
       FLD (PIC'BB(7)-9V.(6)9'),
    4 BLK22 RECORD,
      5 F22 FLD (CHAR 80),
    4 NAMES3 RECORD,
      5 NM3 FLD (CHAR 124),
    4 BLK3 RECORD,
      5 F3 FLD (CHAR 80),
    4 VALUES3(1:10) RECORD,
      5 YEAR3 FLD (PIC '9999'),
```

```
   5 (X$16, X$36, X$46, X$56, X$77,PX$76, X$26)
     FLD (PIC'BB(7)-9V,(6)9')
4 BLK33 RECORD,
  5 F33 FLD (CHAR 80),
4 NAME$4 RECORD,
  5 NM4 FLD (CHAR 124),
4 BLK4 RECORD,
  5 F4 FLD (CHAR 80),
4 VALUES4(1:10) RECORD,
  5 YEAR4 FLD (PIC '9999'),
  5 (X$76,PX$75,X$45, X$25, X$15, X$75,PX$74)
    FLD (PIC'BB(7)-9V,(6)9'),
4 BLK44 RECORD,
  5 F44 FLD (CHAR 80),
4 NAME5 RECORD,
  5 NM5 FLD (CHAR 124),
4 BLK5 RECORD,
  5 F5 FLD (CHAR 80),
4 VALUES5(1:10) RECORD,
  5 YEAR5 FLD (PIC '9999'),
  5 (X$24, X$14, X$74,PX$73, X$43, X$23, X$73)
    FLD (PIC'BB(7)-9V,(6)9'),
4 BLK55 RECORD,
  5 F55 FLD (CHAR 80),
4 NAME6 RECORD,
  5 NM6 FLD (CHAR 124),
4 BLK6 RECORD,
  5 F6 FLD (CHAR 80),
4 VALUES6(1:10) RECORD,
  5 YEAR6 FLD (PIC '9999'),
  5 (PX$72, X$62, X$42, X$72, X$71, X$7, PX$71)
    FLD (PIC'BB(7)-9V,(6)9'),
4 BLK66 RECORD,
  5 F66 FLD (CHAR 80),
4 NAME7 RECORD,
  5 NM7 FLD (CHAR 124),
4 BLK7 RECORD,
  5 F7 FLD (CHAR 80),
4 VALUES7(1:10) RECORD,
  5 YEAR7 FLD (PIC '9999'),
  5 (X$41, X$21, X$17, X$27, X$37, X$47, X$57)
     FLD (PIC'BB(7)-9V,(6)9'),
4 BLK77 RECORD,
  5 F77 FLD (CHAR 80),
4 NAME8 RECORD,
  5 NM8 FLD (CHAR 124),
4 BLK8 RECORD,
  5 F8 FLD (CHAR 80),
4 VALUES8(1:10) RECORD,
  5 YEAR8 FLD (PIC '9999'),
  5 (X$67,PX$77, M$7, PM$7)
     FLD (PIC'BB(7)-9V,(6)9'),
4 BLK88 RECORD,
  5 F88 FLD (CHAR 80),
4 EXD_HD RECORD,
  5 EXD_HDN FLD (CHAR 124),
4 EXD_FR RECORD,
  5 EXD_FF FLD (CHAR 80),
4 NAME9 RECORD,
  5 NM9 FLD (CHAR 124),
4 BLK9 RECORD,
  5 F9 FLD (CHAR 80),
4 VALUES9(1:10) RECORD,
  5 YEAR9 FLD (PIC '9999'),
  5 (0M$2  ,0PX$2 ,0X$2  ,0USAPY,0USAY,0USAR ,0M$1)
     FLD (PIC'BB(7)-9V,(6)9'),
```

```
    4 BLK99 RECORD,
      5 F99 FLD (CHAR 80),
    4 NAME10 RECORD,
      5 NM10 FLD (CHAR 124),
    4 BLK10 RECORD,
      5 F10 FLD (CHAR 80),
    4 VALUES10(1:10) RECORD,
      5 YEAR10 FLD (PIC '9999'),
      5 (OPY$1 ,OX$1  ,OJAPPY,
         OJAPY ,OJAPR ,OM$3  ,OPX$3)
         FLD (PIC'BB(7)-9V.(6)9'),
    4 BLK1010 RECORD,
      5 F1010 FLD (CHAR 80),
    4 NAME11 RECORD,
      5 NM11 FLD (CHAR 124),
    4 BLK111 RECORD,
      5 F111 FLD (CHAR 80),
    4 VALUES11(1:10) RECORD,
      5 YEAR11 FLD (PIC '9999'),
      5 (OX$3  , OTWNPY,OTWNY ,OTWNR ,OM$4  ,OPY$4 , OX$4 )
         FLD (PIC'BB(7)-9V.(6)9'),
    4 BLK1111 RECORD,
      5 F1111 FLD (CHAR 80),
    4 NAME12 RECORD,
      5 NM12 FLD (CHAR 124),
    4 BLK12 RECORD,
      5 F12 FLD (CHAR 80),
    4 VALUES12(1:10) RECORD,
      5 YEAR12 FLD (PIC '9999'),
      5 (OKRAPY,OKRAY ,OKRAR ,OM$5  ,
         OPX$5 ,OX$5  ,OPHIPY)
         FLD (PIC'BB(7)-9V.(6)9'),
    4 BLK1212 RECORD,
      5 F1212 FLD (CHAR 80),
    4 NAME13 RECORD,
      5 NM13 FLD (CHAR 124),
    4 BLK13 RECORD,
      5 F13 FLD (CHAR 80),
    4 VALUES13(1:10) RECORD,
      5 YEAR13 FLD (PIC '9999'),
      5 (OPHIY ,OPHIR ,OM$6  ,OPX$6 ,OX$6  ,OTHIPY,OTHIY)
         FLD (PIC'BB(7)-9V.(6)9'),
    4 BLK1313 RECORD,
      5 F1313 FLD (CHAR 80),
    4 NAME14 RECORD,
      5 NM14 FLD (CHAR 124),
    4 BLK14 RECORD,
      5 F14 FLD (CHAR 80),
    4 VALUES14(1:10) RECORD,
      5 YEAR14 FLD (PIC '9999'),
      5 OTHIR
         FLD (PIC'BB(7)-9V.(6)9');

END_HDN=COPY(' ',42)!!'E N D O G E N O U S    V A R I A B L E S';
EXD_HDN=COPY(' ',42)!!'E X O G E N O U S     V A R I A B L E S';
(END.VALUES1(T),END.VALUES2(T),END.VALUES3(T),END.VALUES4(T),
 END.VALUES5(T),END.VALUES6(T),END.VALUES7(T),END.VALUES8(T))=T=SIM_PD;
(END.VALUES9(T),END.VALUES10(T),END.VALUES11(T),END.VALUES12(T),
 END.VALUES13(T),END.VALUES14(T))=T=SIM_PD;

(YEAR1(T),YEAR2(T),YEAR3(T),YEAR4(T),YEAR5(T),YEAR6(T),YEAR7(T),YEAR8(T),
 YEAR9(T),YEAR10(T),YEAR11(T),YEAR12(T),YEAR13(T),YEAR14(T))
   =START_YR+T;
(F0,F1,F2,F3,F4,F5,F6,F7,F8,F11,F22,F33,F44,F55,F66,F77)=' ';
(F9,F10,F111,F12,F13,F14,F88,F99,F1010,F1111,F1212,F1313)=' ';
(END_FF,EXD_FF)=' ';
```

```
FHD=COPY('   .44)!!'LOCAL SIMULATION REPORT FOR: WORLD2';
NM1='YEAR       X$31          X$51          X$61          X$12'
     !!'          X$32          X$52          X$13';
NM2='YEAR       X$53          X$63          X$34          X$54'
     !!'          X$64          X$35          X$65';
NM3='YEAR       X$16          X$36          X$46          X$56'
     !!'          X$77          PX$76         X$26';
NM4='YEAR       X$76          PX$75         X$45          X$25'
     !!'          X$15          X$75          PX$74';
NM5='YEAR       X$24          X$14          X$74          PX$73'
     !!'          X$43          X$23          X$73';
NM6='YEAR       PX$72         X$62          X$42          X$72'
     !!'          X$71          X$7           PX$71';
NM7='YEAR       X$41          X$21          X$17          X$27'
     !!'          X$37          X$47          X$57';
NM8='YEAR       X$67          PX$77         M$7           PM$7';

NM9='YEAR       M$2           PX$2          X$2           USAPY'
     !!'          USAY          USAR          M$1';

NM10='YEAR      PX$1          X$1           JAPPY         JAPY'
     !!'               JAPR          M$3           PX$3';
NM11='YEAR      X$3           TWNPY         TWNY          TWNR'
     !!'               M$4           PX$4          X$4';
NM12='YEAR      KRAPY         KRAY          KRAR          M$5'
     !!'               PX$5          X$5           PHILPY';
NM13='YEAR      PHIY          PHIR          M$6           PX$6'
     !!'               X$6           THIPY         THIY';
NM14='YEAR      THIR';


OM$2=M$2;
OPX$2=PX$2;
OX$2=X$2;
OUSAPY=USAPY;
OUSAY=USAY;
OUSAR=USAR;
OM$1=M$1;
OPX$1=PX$1;
OX$1=X$1;
OJAPPY=JAPPY;
OJAPY=JAPY;
OJAPR=JAPR;
OM$3=M$3;
OPX$3=PX$3;
OX$3=X$3;
OTWNPY=TWNPY;
OTWNY=TWNY;
OTWNR=TWNR;
OM$4=M$4;
OPX$4=PX$4;
OX$4=X$4;
OKRAPY=KRAPY;
OKRAY=KRAY;
OKRAR=KRAR;
OM$5=M$5;
OPX$5=PX$5;
OX$5=X$5;
OPHIPY=PHIPY;
OPHIY=PHIY;
OPHIR=PHIR;
OM$6=M$6;
OPX$6=PX$6;
OX$6=X$6;
OTHIPY=THIPY;
OTHIY=THIY;
```

```
QTHIR=THIR;

BLOCK WRD: MAX ITER IS 100, RELATIVE ERROR IS 0.01;

(USARC,JAPRC,TWNRC,PHIRC,TAIRC,KRARC)=DEPENDS_ON(QIG);

INITIAL.M$1(T)=IF T=1 THEN TS_DATA(7,T)
               ELSE M$1(T-1);
INITIAL.PX$1(T)=IF T=1 THEN TS_DATA(8,T)
                ELSE PX$1(T-1);
INITIAL.X$1(T)=IF T=1 THEN TS_DATA(9,T)
               ELSE X$1(T-1);
INITIAL.JAPPY(T)=IF T=1 THEN TS_DATA(10,T)
                 ELSE JAPPY(T-1);
INITIAL.JAPY(T)=IF T=1 THEN TS_DATA(11,T)
                ELSE JAPY(T-1);
INITIAL.JAPR(T)=IF T=1 THEN TS_DATA(12,T)
                ELSE JAPR(T-1);

INITIAL.M$2(T)=IF T=1 THEN TS_DATA(1,T)
               ELSE M$2(T-1);
INITIAL.PX$2(T)=IF T=1 THEN TS_DATA(2,T)
                ELSE PX$2(T-1);
INITIAL.X$2(T)=IF T=1 THEN TS_DATA(3,T)
               ELSE X$2(T-1);
INITIAL.USAPY(T)=IF T=1 THEN TS_DATA(4,T)
                 ELSE USAPY(T-1);
INITIAL.USAY(T)=IF T=1 THEN TS_DATA(5,T)
                ELSE USAY(T-1);
INITIAL.USAR(T)=IF T=1 THEN TS_DATA(6,T)
                ELSE USAR(T-1);

INITIAL.M$3(T)=IF T=1 THEN TS_DATA(13,T)
               ELSE M$3(T-1);
INITIAL.PX$3(T)=IF T=1 THEN TS_DATA(14,T)
                ELSE PX$3(T-1);
INITIAL.X$3(T)=IF T=1 THEN TS_DATA(15,T)
               ELSE X$3(T-1);
INITIAL.TWNPY(T)=IF T=1 THEN TS_DATA(16,T)
                 ELSE TWNPY(T-1);
INITIAL.TWNY(T)=IF T=1 THEN TS_DATA(17,T)
                ELSE TWNY(T-1);
INITIAL.TWNR(T)=IF T=1 THEN TS_DATA(18,T)
                ELSE TWNR(T-1);

INITIAL.M$4(T)=IF T=1 THEN TS_DATA(19,T)
               ELSE M$4(T-1);
INITIAL.PX$4(T)=IF T=1 THEN TS_DATA(20,T)
                ELSE PX$4(T-1);
INITIAL.X$4(T)=IF T=1 THEN TS_DATA(21,T)
               ELSE X$4(T-1);
INITIAL.KRAPY(T)=IF T=1 THEN TS_DATA(22,T)
                 ELSE KRAPY(T-1);
INITIAL.KRAY(T)=IF T=1 THEN TS_DATA(23,T)
                ELSE KRAY(T-1);
INITIAL.KRAR(T)=IF T=1 THEN TS_DATA(24,T)
                ELSE KRAR(T-1);

INITIAL.M$5(T)=IF T=1 THEN TS_DATA(25,T)
               ELSE M$5(T-1);
INITIAL.PX$5(T)=IF T=1 THEN TS_DATA(26,T)
                ELSE PX$5(T-1);
INITIAL.X$5(T)=IF T=1 THEN TS_DATA(27,T)
               ELSE X$5(T-1);
INITIAL.PHIPY(T)=IF T=1 THEN TS_DATA(28,T)
                 ELSE PHIPY(T-1);
```

```
INITIAL.PHIY(T)=IF T=1 THEN TS_DATA(29,T)
                ELSE PHIY(T-1);
INITIAL.PHIR(T)=IF T=1 THEN TS_DATA(30,T)
                ELSE PHIR(T-1);

INITIAL.M$6(T)=IF T=1 THEN TS_DATA(31,T)
               ELSE M$6(T-1);
INITIAL.PX$6(T)=IF T=1 THEN TS_DATA(32,T)
                ELSE PX$6(T-1);
INITIAL.X$6(T)=IF T=1 THEN TS_DATA(33,T)
               ELSE X$6(T-1);
INITIAL.THIPY(T)=IF T=1 THEN TS_DATA(34,T)
                 ELSE THIPY(T-1);
INITIAL.THIY(T)=IF T=1 THEN TS_DATA(35,T)
                ELSE THIY(T-1);
INITIAL.THIR(T)=IF T=1 THEN TS_DATA(36,T)
                ELSE THIR(T-1);


/* THE FOLLOWING EQUATIONS ARE PROVIDED BY MR. YASUDA, LINK PROJECT, 1984 */

BLOCK WORLDMD: MAX ITER IS 100, RELATIVE ERROR IS 0.01;


 X$31(T)  = IF TDLAG
            THEN -95.3078 + 0.001490 * JAPY(T)*1.001*1000/357.60
                 + 0.3594 * X$31(T-1)
            ELSE TS_DATA(37,T);

 X$51(T)  = IF TDLAG
            THEN 219.2138 + 0.001507 * JAPY(T)*1.001*1000.0/357.60
                 - 127.4736 * PX$5(T)/(JAPPY(T)/1.001*357.60/JAPR(T))
            ELSE TS_DATA(56,T);

 X$61(T)  = IF TDLAG
            THEN -10.5131 + 0.0005825 * JAPY(T)*1.001*1000.0/357.60
                 + 0.5160 * X$61(T-1)
            ELSE TS_DATA(38,T);

 X$12(T)  = IF TDLAG
            THEN 1386.7429 + 0.003833 * USAY(T)*0.9136*1000.0 +
                 0.7382 * X$12(T-1)
                 - 3132.5297 * PX$1(T)/(USAPY(T)/0.9136/USAR(T))
            ELSE TS_DATA(39,T);

 X$32(T)  = IF TDLAG
            THEN -59.5317 + 0.0005630 * USAY(T)*0.9136*1000.0 +
                 0.8706 * X$32(T-1)
                 - 299.5960 * PX$3(T)/(USAPY(T)/0.9136/USAR(T))
            ELSE TS_DATA(40,T);

 X$52(T)  = IF TDLAG
            THEN 169.3380 + 0.00002769 * USAY(T)*0.9136*1000.0 +
                 0.7609 * X$52(T-1)
                 - 83.9859 * PX$5(T)/(USAPY(T)/0.9136/USAR(T))
            ELSE TS_DATA(41,T);

 ($13(T)  = IF TDLAG
            THEN 29.3277 + 0.1295 * TWNY(T)*0.967/40.10 +
                 0.5790 * X$13(T-1)
                 - 386.9854 * PX$1(T)*TWNR(T)/40.10
            ELSE TS_DATA(42,T);

 X$53(T)  = IF TDLAG
            THEN 10.2835 + 0.003610 * TWNY(T)*0.967/40.10 +
                 0.2533 * X$53(T-1)
```

```
            - 15.8030 * PX$5(T)*TWNR(T)/40.10
          ELSE TS_DATA(43,T);

X$63(T)   = IF T>LAG
          THEN -16.4982 + 0.007688 * TWNY(T)*0.967/40.10
          ELSE TS_DATA(57,T);

X$34(T)   =  IF T>LAG
           THEN -28.0751 + 0.007392 * KRAY(T)*1000.0/316.0
           ELSE TS_DATA(58,T);

X$54(T)   = IF T>LAG
          THEN 9.0581 + 0.002960 * KRAY(T)*1000.0/316.0 -
               11.5755 * PX$5(T)*KRAR(T)/316.0
          ELSE TS_DATA(59,T);

X$64(T)   = IF T>LAG
          THEN -2.7504 + 0.0005176 * KRAY(T)*1000.0/316.0 +
               1.0009 * X$64(T-1)
          ELSE TS_DATA(44,T);

X$35(T)   = IF T>LAG
          THEN -12.3120 + 0.004234 * PHIY(T)*1.257/5.729 +
               0.7258 * X$35(T-1)
               - 11.1044 * PX$3(T)/ (PHIPY(T)/1.257*5.729/PHIR(T))
          ELSE TS_DATA(45,T);

X$65(T)   = IF T>LAG
          THEN 8.1989 + 0.003371 * PHIY(T)*1.257/5.729 -
               20.3589 * PX$6(T)
               / (PHIPY(T) / 1.257*5.729/PHIR(T))
          ELSE TS_DATA(60,T);

X$16(T)   = IF T>LAG
          THEN 372.8650 + 0.05599 * THIY(T)*1.135/20.930 +
               0.3090 * X$16(T-1)
               - 429.7248 * PX$1(T)/
               (THIPY(T)/1.135*20.93/THIR(T))
          ELSE TS_DATA(46,T);

X$36(T)   = IF T>LAG
          THEN 4.9402 + 0.005841 * THIY(T)*1.135/20.930 -
               12.5028 * PX$3(T)
              /(THIPY(T)/1.135*20.93/THIR(T))
          ELSE TS_DATA(61,T);

X$46(T)   = IF T>LAG
          THEN 0.6136 + 0.001597 * THIY(T)*1.135/20.930 +
               0.2952 * X$46(T-1)
               - 5.8554 * PX$4(T)/(THIPY(T)/1.135*20.930/THIR(T))
          ELSE TS_DATA(47,T);

X$56(T)   = IF T>LAG
          THEN 2.1872 + 0.0003950 * THIY(T)*1.135/20.930 +
               0.4745 * X$56(T-1)
               - 3.6961 * PX$5(T)/(THIPY(T)/1.135*20.93/THIR(T))
          ELSE TS_DATA(48,T);

WT$(T)    = IF T>LAG
          THEN X$1(T) + X$2(T) + X$3(T) +
               X$4(T) + X$5(T) + X$6(T) + X$7(T)
          ELSE TS_DATA(62,T);

X$77(T)   = IF T>LAG
          THEN ALPHA(T) * WT$(T)
          ELSE TS_DATA(63,T);
```

```
PX$76(T) = IF TDLAG
           THEN BETA76(T) * PW$(T)
           ELSE TS_DATA(64,T);

PM$6(T)  = IF TDLAG
           THEN (PX$1(T) * X$16(T) +
                PX$2(T) * X$26(T) +
                PX$3(T) * X$36(T) +
                PX$4(T) * X$46(T) +
                PX$5(T) * X$56(T) +
                PX$6(T) * X$66(T) +
                PX$76(T)* X$76(T)) /
                (X$16(T) + X$26(T) + X$36(T) + X$46(T) +
                 X$56(T) + X$66(T) + X$76(T))
           ELSE TS_DATA(65,T);

X$26(T)  = IF TDLAG
           THEN 401.3615 + 0.01491 * THIY(T)*1.135/20.930 +
                0.2443 * X$26(T-1)
                - 137.6322 * PX$2(T)/PM$6(T) -
                245.0900 * PX$2(T)/(THIPY(T)/1.135*20.930/THIR(T))
           ELSE TS_DATA(49,T);

X$76(T)  = IF TDLAG
           THEN M$6(T)-(X$16(T)+X$26(T)+
                X$36(T)+X$46(T)+X$56(T)+X$66(T))
           ELSE TS_DATA(66,T);

PX$75(T) = IF TDLAG
           THEN BETA75(T) * PW$(T)
           ELSE TS_DATA(67,T);

X$45(T)  = IF TDLAG
           THEN 2.0984 + 0.001067 * PHIY(T)*1.257/5.729 -
                7.4456 * PX$4(T)/PM$5(T)
           ELSE TS_DATA(68,T);

X$25(T)  = IF TDLAG
           THEN 735.0191 + 0.01530 * PHIY(T)*1.257/5.729 +
                0.3397 * X$25(T-1)
                - 361.9462 * PX$2(T)/PM$5(T) -
                260.8525 * PX$2(T)/(PHIPY(T)/1.257*5.729/PHIR(T))
           ELSE TS_DATA(50,T);

PM$5(T)  = IF TDLAG
           THEN (PX$1(T) * X$15(T) + PX$2(T) * X$25(T) +
                 PX$3(T) * X$35(T) + PX$4(T) * X$45(T) +
                 PX$5(T) * X$55(T) + PX$6(T) * X$65(T) +
                 PX$75(T)* X$75(T)) /
                (X$15(T) + X$25(T) + X$35(T) +
                 X$45(T) + X$55(T) + X$65(T) + X$75(T))
           ELSE TS_DATA(69,T);

X$15(T)  = IF TDLAG
           THEN 983.1558+0.1030 * PHIY(T)*1.257/5.729+
                0.3630 * X$15(T-1)
                - 1223.8837 * PX$1(T)/PM$5(T) -
                218.5203 * PX$1(T)*PHIR(T)/5.729
           ELSE TS_DATA(51,T);

X$75(T)  = IF TDLAG
           THEN M$5(T) - (X$15(T) + X$25(T) + X$35(T) +
                X$45(T) + X$55(T) + X$65(T))
           ELSE TS_DATA(70,T);
```

```
PX$74(T) = IF T>LAG
           THEN BETA74(T) * PW$(T)
           ELSE TS_DATA(71,T);

X$24(T)  = IF T>LAG
           THEN 813.5893 + 0.07662 * KRAY(T)*1000.0/316.0 -
                505.2523 * PX$2(T)/PM$4(T)
                - 266.6582 * PX$2(T)/(KRAPY(T)*316.0/KRAR(T))
           ELSE TS_DATA(72,T);

PM$4(T)  = IF T>LAG
           THEN (PX$1(T) * X$14(T) + PX$2(T) * X$24(T) +
                 PX$3(T) * X$34(T) + PX$4(T) * X$44(T) +
                 PX$5(T) * X$54(T) + PX$6(T) * X$64(T) +
                 PX$74(T)* X$74(T)) /
                 (X$14(T) + X$24(T) + X$34(T) +
                 X$44(T) + X$54(T) + X$64(T) + X$74(T))
           ELSE TS_DATA(73,T);

X$14(T)  = IF T>LAG
           THEN 3140.8171 + 0.1143 * KRAY(T)*1000.0/316.0 -
                3087.0086 * PX$1(T)/PM$4(T)
                - 165.7276 * PX$1(T)/(KRAPY(T)*316.0/KRAR(T))
           ELSE TS_DATA(74,T);

X$74(T)  = IF T>LAG
           THEN M$4(T) - (X$14(T) + X$24(T) + X$34(T) +
                X$44(T) + X$54(T) + X$64(T))
           ELSE TS_DATA(75,T);

PX$73(T) = IF T>LAG
           THEN BETA73(T) * PW$(T)
           ELSE TS_DATA(76,T);

X$43(T)  = IF T>LAG
           THEN 12.2308 + 0.004332 * TWNY(T)*0.9670/40.10 +
                0.3139 * X$43(T-1)
                - 27.3673 * PX$4(T)/PM$3(T)
           ELSE TS_DATA(52,T);

PM$3(T)  = IF T>LAG
           THEN (PX$1(T) * X$13(T) +
                 PX$2(T) * X$23(T) +
                 PX$3(T) * X$33(T) +
                 PX$4(T) * X$43(T) +
                 PX$5(T) * X$53(T) +
                 PX$6(T) * X$63(T) +
                 PX$73(T) * X$73(T)) /
                (X$13(T) + X$23(T) + X$33(T) +
                 X$43(T) + X$53(T) + X$63(T) + X$73(T))
           ELSE TS_DATA(77,T);

X$23(T) = IF T>LAG
          THEN 1144.7830 + 0.1027 * TWNY(T)*0.9670/40.10 -
               1253.4655 * PX$2(T)/PM$3(T)
          ELSE TS_DATA(78,T);

X$73(T) = IF T>LAG
          THEN M$3(T) -
              (X$13(T) + X$23(T) + X$33(T) +
               X$43(T) + X$53(T) + X$63(T))
          ELSE TS_DATA(79,T);

PX$72(T)= IF T>LAG
          THEN BETA72(T) * PW$(T)
          ELSE TS_DATA(80,T);
```

```
X$62(T) = IF T>LAG
          THEN 92.0317 + 0.0001201 * USAY(T)*0.9136*1000.0 +
               0.2428 * X$62(T-1)
               - 113.0800 * PX$6(T)/PM$2(T) -
               16.4208 * PX$6(T)/(USAPY(T)/0.9136/USAR(T))
          ELSE TS_DATA(53,T);

PM$2(T) = IF T>LAG
          THEN (PX$1(T) * X$12(T) +
                PX$2(T) * X$22(T) +
                PX$3(T) * X$32(T) +
                PX$4(T) * X$42(T) +
                PX$5(T) * X$52(T) +
                PX$6(T) * X$62(T) +
                PX$72(T)* X$72(T)) /
               (X$12(T) + X$22(T) + X$32(T) +
                X$42(T) + X$52(T) + X$62(T) + X$72(T))
           ELSE TS_DATA(81,T);

X$42(T) = IF T>LAG
          THEN -261.4902 + 0.001363 * USAY(T)*0.9136*1000.0 +
               0.4858 * X$42(T-1)
               - 659.1006 * PX$4(T)/PM$2(T) -
               170.9580 * PX$4(T)/(USAPY(T)/0.9136/USAR(T))
          ELSE TS_DATA(54,T);

X$72(T) = IF T>LAG
          THEN M$2(T) - (X$12(T) + X$22(T) +
               X$32(T) + X$42(T) + X$52(T) + X$62(T))
          ELSE TS_DATA(82,T);

X$71(T) = IF T>LAG
          THEN M$1(T) - (X$11(T) + X$21(T) +
               X$31(T) + X$41(T) + X$51(T) + X$61(T))
          ELSE TS_DATA(83,T);

X$7(T)  = IF T>LAG
          THEN X$71(T) + X$72(T) + X$73(T) +
               X$74(T) + X$75(T) + X$76(T) +X$77(T)
          ELSE TS_DATA(84,T);

PW$(T)  = IF T>LAG
          THEN (PX$1(T) * X$1(T) +
                PX$2(T) * X$2(T) +
                PX$3(T) * X$3(T) +
                PX$4(T) * X$4(T) +
                PX$5(T) * X$5(T) +
                PX$6(T) * X$6(T) +
                PX$7(T) * X$7(T)) /
               (X$1(T) + X$2(T) + X$3(T) +
                X$4(T) + X$5(T) + X$6(T) + X$7(T))
           ELSE TS_DATA(85,T);

PX$71(T) = IF T>LAG
           THEN BETA71(T) * PW$(T)
           ELSE TS_DATA(86,T);

X$41(T)  = IF T>LAG
           THEN 210.7135 + 0.003307 * JAPY(T)*1.001*1000.0/357.60
                + 0.2608 * X$41(T-1) - 599.0852 * PX$4(T-1)/PM$1(T)
           ELSE TS_DATA(55,T);

PM$1(T)  = IF T>LAG
           THEN (PX$1(T) * X$11(T) +
                 PX$2(T) * X$21(T) +
                 PX$3(T) * X$31(T) +
```

```
                    PX$4(T) * X$41(T) +
                    PX$5(T) * X$51(T) +
                    PX$6(T) * X$61(T) +
                    PX$71(T)* X$71(T)) /
                   (X$11(T) + X$21(T) + X$31(T) +
                    X$41(T) + X$51(T) + X$61(T) + X$71(T))
              ELSE TS_DATA(87,T);

X$21(T) = IF TDLAG
              THEN 3210.5984 + 0.02082 * JAPY(T)*1.001*1000.0/357.60
                   - 3169.0851 * PX$2(T)/PM$1(T)
              ELSE TS_DATA(88,T);

X$17(T) = IF TDLAG
              THEN X$1(T) - (X$11(T) + X$12(T) +
                   X$13(T) + X$14(T)+X$15(T)+X$16(T))
              ELSE TS_DATA(89,T);

X$27(T) = IF TDLAG
              THEN X$2(T) - (X$21(T) + X$22(T) +
                   X$23(T) + X$24(T)+X$25(T)+X$26(T))
              ELSE TS_DATA(90,T);

X$37(T) = IF TDLAG
              THEN X$3(T) - (X$31(T) + X$32(T) +
                   X$33(T) + X$34(T)+X$35(T)+X$36(T))
              ELSE TS_DATA(91,T);
X$47(T) = IF TDLAG
              THEN X$4(T) - (X$41(T) + X$42(T) +
                   X$43(T) + X$44(T)+X$45(T)+X$46(T))
              ELSE TS_DATA(92,T);

X$57(T) = IF TDLAG
              THEN X$5(T) - (X$51(T) + X$52(T) +
                   X$53(T) + X$54(T)+X$55(T)+X$56(T))
              ELSE TS_DATA(93,T);

X$67(T) = IF TDLAG
              THEN X$6(T) - (X$61(T) + X$62(T) +
                   X$63(T) + X$64(T)+X$65(T)+X$66(T))
              ELSE TS_DATA(94,T);

PX$77(T)= IF TDLAG
              THEN BETA77(T) * PW$(T)
              ELSE TS_DATA(95,T);

M$7(T)  = IF TDLAG
              THEN X$17(T) + X$27(T) + X$37(T) +
                   X$47(T) + X$57(T) + X$67(T)+X$77(T)
              ELSE TS_DATA(96,T);

PM$7(T) = IF TDLAG
              THEN (PX$1(T) * X$17(T) +
                    PX$2(T) * X$27(T) +
                    PX$3(T) * X$37(T) +
                    PX$4(T) * X$47(T) +
                    PX$5(T) * X$57(T) +
                    PX$6(T) * X$67(T) +
                    PX$77(T)* X$77(T)) /
                   (X$17(T) + X$27(T) + X$37(T) +
                    X$47(T) + X$57(T) + X$67(T) + X$77(T))
              ELSE TS_DATA(97,T);

END WORLDMD;

/* DEFINE ADDRESS OF THE MESSAGE TO BE SENT */
```

```
ADR(T,J)=IF J=1 THEN CHPTRLB('I1JAPANS_MBX')
         ELSE IF J=2 THEN 'I1USAS_MBX'
         ELSE IF J=3 THEN 'I1TAIWANS_MBX'
         ELSE IF J=4 THEN 'I1KOREAS_MBX'
         ELSE IF J=5 THEN 'I1PHILS_MBX'
         ELSE 'I1THAIS_MBX';

/* DEFINE THE MESSAGE */
PM$Z(T,J)=IF J=1 THEN PM$1(T)
          ELSE IF J=2 THEN PM$2(T)
          ELSE IF J=3 THEN PM$3(T)
          ELSE IF J=4 THEN PM$4(T)
          ELSE IF J=5 THEN PM$5(T)
          ELSE PM$6(T);
QWT$(T,J)=WT$(T);
QPW$(T,J)=PW$(T);
END WRD;
```

A2.3 THE USA MODULE

```
MODULE: USA;
SOURCE: I1USA,TSDUSA, CTRUSA;
TARGET: O1USA,O2USA;

1 I1USA FILE ORG MAIL,  /* RECEIVED FROM THE WORLD MODULE */
  2 I1R(*) RECORD,
    3 PROC_ID FLD (CHAR 14),
    3 (WT$,PW$,PM$2)
      ARE FLD (DEC FLOAT(15));

 END.I1R(T)=T=SIM_PD;

1 TSDUSA FILE,
  2 IR(1:18) GRP,
    3 HDR RECORD,
      4 HDD FLD (CHAR 21),
    3 TR(10) RECORD,
      4 TS_DATA FLD (PIC'BB(10)-9V.(7)9');

1 CTRUSA FILE, /* THE CONTROL FILE */
  2 CR RECORD,
    3 START_YR FLD (PIC '9999'), /* STARTING YEAR OF SIMULATION */
    3 LAG      FLD (PIC '9'),    /* LAG FROM THE STARTING YEAR */
    3 SIM_PD   FLD (PIC '99'),   /* SIMULATION PERIOD.         */
  2 I21R RECORD,
    3 DHD FLD (CHAR 109),
  2 I2R(*) RECORD,
    3 HDYR FLD (CHAR 4),
    3 (USAGP,XS$2,MS$2,DUM,USAR)
      ARE FLD (PIC'BB(10)-9V.(7)9');

1 O1USA FILE ORG MAIL,
  2 O1R(*) RECORD,  /* SEND TO THE WORLD MODULE */
    3 MAIL_ADR FLD (CHAR 10),
    3 (M$2,PX$2,X$2,USAPY,USAY,USAR)
      ARE FLD (DEC FLOAT(15));

MAIL_ADR='I1USAS';

1 O2USA FILE,  /* LOCAL RESULTS */
  3 HD RECORD,
    4 HFD FLD (CHAR 132),
  3 BLKO RECORD,
    4 FO FLD (CHAR 132),
  3 VALUES GRP,
    4 END_HD RECORD,
      5 END_HDD FLD (CHAR 124),
    4 BKK1 RECORD,
      5 BKF1 FLD (CHAR 80),
    4 NAMES1 RECORD,
      5 NM1 FLD (CHAR 132),
    4 BLK1 RECORD,
      5 F1 FLD (CHAR 132),
    4 VALUES1(1:10) RECORD,
      5 YEAR1 FLD (PIC '9999'),
      5 (OM$2,OPX$2,OX$2,OUSAPY,OUSAY,USAM,USAPX)
        ARE  FLD(PIC'BB(7)-9V.(6)9'),
    4 BKK2 RECORD,
      5 BKF2 FLD (CHAR 80),
    4 NAMES2 RECORD,
      5 NM2 FLD (CHAR 132),
    4 BLK2 RECORD,
      5 F2 FLD (CHAR 132),
    4 VALUES2(1:10) RECORD,
```

```
      5 YEAR2 FLD (PIC '9999'),
      5 (USAX,USAI,USAC)
        ARE  FLD(PIC'BB(7)-9V.(6)9'),
    4 BKK3 RECORD,
      5 BKF3 FLD (CHAR 80),
    4 EXD_HD RECORD,
      5 EXD_HDD FLD (CHAR 124),
    4 EXD_B RECORD,
      5 EXD_F FLD (CHAR 124),
    4 NAMES3 RECORD,
      5 NM3 FLD (CHAR 132),
    4 BLK3 RECORD,
      5 F3 FLD (CHAR 132),
    4 VALUES3(1:10) RECORD,
      5 YEAR3 FLD (PIC '9999'),
      5 (OWT$,OPW$,OPM$2,OUSAR,OUSAGP,OXS$2,OMS$2)
        ARE  FLD(PIC'BB(7)-9V.(6)9'),
    4 BKK4 RECORD,
      5 BKF4 FLD (CHAR 80),
    4 NAMES4 RECORD,
      5 NM4 FLD (CHAR 132),
    4 BLK4 RECORD,
      5 F4 FLD (CHAR 132),
    4 VALUES4(1:10) RECORD,
      5 YEAR4 FLD (PIC '9999'),
      5 (ODUM)
        ARE  FLD(PIC'BB(7)-9V.(6)9');

END_HDD=COPY(' ',42)!!'E N D O G E N O U S   V A R I A B L E S';
EXD_HDD=COPY(' ',42)!!'E X O G E N O U S    V A R I A B L E S';
(END.VALUES1(T),END.VALUES2(T),END.VALUES3(T),END.VALUES4(T))=T=SIM_PD;

OWT$=WT$;
OPW$=PW$;
OPM$2=PM$2;
OM$2=M$2;
OPX$2=PX$2;
OX$2=X$2;
OUSAPY=USAPY;
OUSAY=USAY;
OUSAGP=USAGP;
OXS$2=XS$2;
OMS$2=MS$2;
ODUM=DUM;
OUSAR=CTRUSA.USAR;

(YEAR1(T),YEAR2(T),YEAR3(T),YEAR4(T))=START_YR+T;
(F0,F1,F2,F3,F4,BKF1,BKF2,BKF3,BKF4,EXD_F)=' ';

HFD=COPY(' ',46)!!'LOCAL SIMULATION REPORT FOR: USA';

NM1='YEAR       M$2      '!!'       PX$2     '
      !!'        X$2     '!!'       USAPY    '
      !!'        USAY    '!!'       USAM     '
      !!'       USAPX    ';
NM2='YEAR       USAX     '!!'       USAI     '
      !!'       USAC     ';

NM3='YEAR        WT$     '!!'       PW$      '
      !!'        PM$2    '!!'       USAR     '
      !!'        USAGP   '!!'       XS$2     '
      !!'        MS$2    ';
NM4='YEAR        DUM';

T SUBSCRIPT;
```

```
BLOCK US: MAX ITER IS 20, RELATIVE ERROR IS 0.01;

I1R(T)=DEPENDS_ON(O1R(T));

INITIAL.WT$(T)=IF T<=LAG THEN TS_DATA(9,T)
               ELSE WT$(T-1);
INITIAL.PW$(T)=IF T<=LAG THEN TS_DATA(10,T)
               ELSE PW$(T-1);
INITIAL.PM$2(T)=IF T<=LAG THEN TS_DATA(11,T)
               ELSE PM$2(T-1);


/* THE FOLLOWING EQUATIONS ARE PROVIDED BY MR. YASUDA, LINK PROJECT, 1984 */

BLOCK USAMD: MAX ITER IS 20, RELATIVE ERROR IS 0.01;

 M$2(T)   = IF T>LAG
            THEN -27696.3164 + 0.02479 * USAY(T)*0.9136*1000 +
                 0.7377 * M$2(T-1) +
                 16213.6127 * USAPY(T)/0.9136/PM$2(T)*CTRUSA.USAR(T)
            ELSE TS_DATA(6,T);

 USAM(T)  = IF T>LAG
            THEN (M$2(T) + MS$2(T))/ (0.891 * 1000.)
            ELSE TS_DATA(12,T);

 USAPY(T) = IF T>LAG
            THEN 0.03639 + 0.6478 * USAY(T)/1171.1 +
                 0.2820 * PM$2(T)/1.121*CTRUSA.USAR(T)
            ELSE TS_DATA(13,T);

 USAPX(T) = IF T>LAG
            THEN 0.03325 + 0.3617 * USAPY(T) + 0.6260 * PW$(T)
                 *CTRUSA.USAR(T)/1.128
            ELSE TS_DATA(14,T);

 PX$2(T)  = IF T>LAG
            THEN -0.007977 + 1.0014 * USAPX(T)*CTRUSA.USAR(T)/0.931
            ELSE TS_DATA(15,T);

 X$2(T)   = IF T>LAG
            THEN 36598.1049 + 0.08361 * WT$(T) + 0.2892 * X$2(T-1)
                 - 32236.8591 * PX$2(T)/PW$(T)
            ELSE TS_DATA(7,T);

 USAX(T)  = IF T>LAG
            THEN (X$2(T) + XS$2(T))/ (9.31 * 1000.)
            ELSE TS_DATA(16,T);

 USAI(T)  = IF T>LAG
            THEN -33.6486 + 0.1879 * USAY(T) - 31.1877 * DUM(T)
            ELSE TS_DATA(17,T);

 USAC(T)  = IF T>LAG
            THEN -22.0848 + 0.2736 * USAY(T) + 0.6127 * USAC(T-1)-
                 9.4185 * DUM(T)
            ELSE TS_DATA(8,T);

 USAY(T)  = IF T>LAG
            THEN USAC(T) + USAI(T) + USAGP(T) + USAX(T) - USAM(T)
            ELSE TS_DATA(18,T);

END USAMD;
END US;
```

A2.4 THE JAPAN MODULE

```
MODULE: JAPAN;
SOURCE: I1JAPAN, TSDJAPAN, CTRJAPAN;
TARGET: O1JAPAN,O2JAPAN;

1 I1JAPAN FILE ORG MAIL, /* SENT FROM THE WORLD MODULE */
  2 I1R(*) RECORD,
    3 PROC_ID FLD (CHAR 14), /* ADDRESS SENT FROM THE WORLD */
    3 (WT$,PW$,PM$1)
      ARE FLD (DEC FLOAT(15));

 END.I1R(T)=T=SIM_PD;


1 CTRJAPAN FILE, /* THE CONTROL FILE */
  2 CR RECORD,
    3 START_YR FLD (PIC '9999'), /* STARTING YEAR OF SIMULATION */
    3 LAG      FLD (PIC '9'),    /* LAG FROM THE STARTING YEAR */
    3 SIM_PD   FLD (PIC '99'),   /* SIMULATION PERIOD.         */
  2 I21R RECORD,
    3 DHD FLD (CHAR 109),
  2 I2R(*) RECORD,               /* LOCAL HISTORICAL DATABASE  */
    3 HDYR FLD (CHAR 4),
    3 (JAPGP,XS$1,MS$1,DUM,JAPR)
      ARE FLD (PIC'BB(10)-9V.(7)9');

1 TSDJAPAN FILE,
  2 IR(1:18) GRP,
    3 HDR RECORD,
      4 HDD FLD (CHAR 21),
    3 TR(10) RECORD,
      4 TS_DATA FLD (PIC'BB(10)-9V.(7)9');



1 O1JAPAN FILE ORG MAIL, /* SEND TO WORLD MODULE */
  2 O1R(*) RECORD,
    3 MAIL_ADR FLD (CHAR 10), /* GIVING WORLD MODULE THE RETURN ADR */
    3 (M$1,PX$1,X$1,JAPPY,JAPY,JAPR)
      ARE FLD (DEC FLOAT(15));

MAIL_ADR='I1JAPANS';

1 O2JAPAN FILE,  /* LOCAL RESULTS */
  3 HD RECORD,
    4 HFD FLD (CHAR 132),
  3 BLK0 RECORD,
    4 F0 FLD (CHAR 80),
  3 VALUES GRP,
    4 END_HD RECORD,
      5 END_HDD FLD (CHAR 124),
    4 BKK1 RECORD,
      5 BKF1 FLD (CHAR 80),
    4 NAMES1 RECORD,
      5 NM1 FLD (CHAR 132),
    4 BLK1 RECORD,
      5 F1 FLD (CHAR 132),
    4 VALUES1(1:10) RECORD,
      5 YEAR1 FLD (PIC '9999'),
      5 (OM$1,OPX$1,OX$1,OJAPPY,OJAPY,JAPM,JAPPY)
        ARE  FLD(PIC'BB(7)-9V.(6)9'),
    4 BKK2 RECORD,
      5 BKF2 FLD (CHAR 80),
    4 NAMES2 RECORD,
      5 NM2 FLD (CHAR 132),
```

```
   4 BLK2 RECORD,
     5 F2 FLD (CHAR 132),
   4 VALUES2(1:10) RECORD,
     5 YEAR2 FLD (PIC '9999'),
     5 (JAPX,JAPI,JAPC)
        ARE  FLD(PIC'BB(7)-9V.(6)9'),
   4 BKK3 RECORD,
     5 BKF3 FLD (CHAR 80),
   4 EXD_HD RECORD,
     5 EXD_HDD FLD (CHAR 124),
   4 EXD_B RECORD,
     5 EXD_F FLD (CHAR 124),
   4 NAMES3 RECORD,
     5 NM3 FLD (CHAR 132),
   4 BLK3 RECORD,
     5 F3 FLD (CHAR 132),
   4 VALUES3(1:10) RECORD,
     5 YEAR3 FLD (PIC '9999'),
     5 (OWT$,OPW$,OPM$1,OJAPR,OJAPGP,OXS$1,OMS$1)
        ARE  FLD(PIC'BB(7)-9V.(6)9'),
   4 BKK4 RECORD,
     5 BKF4 FLD (CHAR 80),
   4 NAMES4 RECORD,
     5 NM4 FLD (CHAR 132),
   4 BLK4 RECORD,
     5 F4 FLD (CHAR 132),
   4 VALUES4(1:10) RECORD,
     5 YEAR4 FLD (PIC '9999'),
     5 (ODUM)
        ARE  FLD(PIC'BB(7)-9V.(6)9');

END_HDD=COPY(' ',42)!!'E N D O G E N O U S    V A R I A B L E S';
EXD_HDD=COPY(' ',42)!!'E X O G E N O U S     V A R I A B L E S';
(END.VALUES1(T),END.VALUES2(T),END.VALUES3(T),END.VALUES4(T))=T=SIM_PD;
OWT$=WT$;
OPW$=PW$;
OPM$1=PM$1;
OM$1=M$1;
OPX$1=PX$1;
OX$1=X$1;
OJAPPY=JAPPY;
OJAPY=JAPY;
OJAPGP=JAPGP;
OXS$1=XS$1;
OMS$1=MS$1;
ODUM=DUM;
OJAPR=CTRJAPAN.JAPR;

(YEAR1(T),YEAR2(T),YEAR3(T),YEAR4(T))=START_YR+T;
(F0,F1,F2,F3,F4,BKF1,BKF2,BKF3,BKF4,EXD_F)=' ';

HFD=COPY(' ',45)!!'LOCAL SIMULATION REPORT FOR: JAPAN';
NM1='YEAR       M$1      '!!'       PX$1     '
      !!'        X$1      '!!'       JAPPV    '
      !!'        JAPY     '!!'       JAPM     '
      !!'       JAPPX     ';
NM2='YEAR       JAPX     '!!'       JAPI     '
      !!'       JAPC      ';

NM3='YEAR        WT$      '!!'       PW$      '
      !!'        PM$1     '!!'       JAPR     '
      !!'        JAPGP    '!!'       XS$1     '
      !!'        MS$1     ';
NM4='YEAR        DUM';

 T SUBSCRIPT;
```

```
BLOCK JAP: MAX ITER IS 20, RELATIVE ERROR IS 0.01:

I1R(T)=DEPENDS_ON(O1R(T));
INITIAL.WT$(T)=IF T<=LAG THEN TS_DATA(4,T)
               ELSE WT$(T-1);
INITIAL.PW$(T)=IF T<=LAG THEN TS_DATA(5,T)
                ELSE PW$(T-1);
INITIAL.PM$1(T)=IF T<=LAG THEN TS_DATA(6,T)
                ELSE PM$1(T-1);

 /*--------------- THE JAPAN MODEL (FROM: YASUDA, LINK PROJECT ------------*/

BLOCK JAPANMD: MAX ITER IS 20, RELATIVE ERROR IS 0.01:


 M$1(T)   = IF T>LAG
            THEN -2412.7034 + 0.08578 * JAPY(T)*1.001/357.6*1000.
                 + 0.1036 * M$1(T-1)
            ELSE TS_DATA(1,T);



 JAPM(T)  = IF T>LAG
            THEN (M$1(T) + MS$1(T))/1000./ (1. / 357.6)
            ELSE TS_DATA(7,T);

 JAPPY(T) = IF T>LAG
            THEN 0.1437 + 0.5280 * JAPY(T)/70613.3 +
                 0.3490 * PM$1(T)*CTRJAPAN.JAPR(T)/357.6
            ELSE TS_DATA(8,T);

 JAPPX(T) = IF T>LAG
            THEN 0.2756 + 0.3126 * JAPPY(T) +
                 0.4104 * PW$(T)*CTRJAPAN.JAPR(T)/357.6
            ELSE TS_DATA(9,T);

 PX$1(T)  = IF T>LAG
            THEN -0.04544 + 1.0691 * JAPPX(T)/CTRJAPAN.JAPR(T)*357.6
            ELSE TS_DATA(10,T);

 X$1(T)   = IF T>LAG
            THEN 9397.7628 + 0.06519 * WT$(T) + 0.2933 * X$1(T-1)
                 - 15585.5870 * PX$1(T)/PW$(T)
            ELSE TS_DATA(2,T);

 JAPX(T)  = IF T>LAG
            THEN (X$1(T) + XS$1(T))/1000./ (1. / 357.6)
            ELSE TS_DATA(11,T);

 JAPI(T)  = IF T>LAG
            THEN -5141.2522 + 0.4528 * JAPY(T) - 3652.4761 * DUM(T)
            ELSE TS_DATA(12,T);

 JAPC(T)  = IF T>LAG
            THEN 2161.6489 + 0.2382 * JAPY(T) + 0.5245 * JAPC(T-1)
            ELSE TS_DATA(3,T);

 JAPY(T)  = IF T>LAG
            THEN JAPC(T) + JAPI(T) + JAPGP(T) + JAPX(T) - JAPM(T)
            ELSE TS_DATA(13,T);

END JAPANMD;
END JAP;
```

A2.5 THE TAIWAN MODULE

```
MODULE:TAIWAN:
SOURCE: I1TAIWAN,TSDTWN, CTRTWN:
TARGET: O1TAIWAN,O2TAIWAN:

1 I1TAIWAN FILE ORG MAIL,
  2 I1R(*) RECORD,
    3 PROC_ID FLD (CHAR 14),
    3 (WT$,PW$,PM$3)
       ARE FLD (DEC FLOAT(15)):

END.I1R(T)=T=SIM_PD;

1 TSDTWN FILE,
  2 IR(1:18) GRP,
    3 HDR RECORD,
      4 HDD FLD (CHAR 21),
    3 TR(10) RECORD,
      4 TS_DATA FLD (PIC'BB(10)-9V.(7)9'):

1 CTRTWN FILE, /* THE CONTROL FILE */
  2 CR RECORD,
    3 START_YR FLD (PIC '9999'), /* STARTING YEAR OF SIMULATION */
    3 LAG      FLD (PIC '9'),    /* LAG FROM THE STARTING YEAR */
    3 SIM_PD   FLD (PIC '99'),   /* SIMULATION PERIOD.         */
  2 I21R RECORD,
    3 DHD FLD (CHAR 88),
  2 I2R(*) RECORD,               /* LOCAL DATABASE             */
    3 HDYR FLD (CHAR 4),
    3 (TWNGP,XS$3,MS$3,TWNR)
       ARE FLD (PIC'BB(10)-9V.(7)9');

1 O1TAIWAN FILE ORG MAIL,
  2 O1R(*) RECORD,
    3 MAIL_ADR FLD (CHAR 10),
    3 (M$3,PX$3,X$3,TWNPY,TWNY,TWNR)
       ARE FLD (DEC FLOAT(15));

MAIL_ADR='I1TAIWANS';

1 O2TAIWAN FILE,  /* LOCAL RESULTS */
  3 HD RECORD,
    4 HFD FLD (CHAR 132),
  3 BLK0 RECORD,
    4 F0 FLD (CHAR 80),
  3 VALUES GRP,
    4 END_HD RECORD,
      5 END_HDD FLD (CHAR 124),
    4 BKK1 RECORD,
      5 BKF1 FLD (CHAR 80),
    4 NAMES1 RECORD,
      5 NM1 FLD (CHAR 132),
    4 BLK1 RECORD,
      5 F1 FLD (CHAR 132),
    4 VALUES1(1:10) RECORD,
      5 YEAR1 FLD (PIC '9999'),
      5 (OM$3,OPX$3,OX$3,OTWNPY,OTWNY,TWNM,TWNPX)
         ARE  FLD(PIC'BB(7)-9V.(6)9'),
    4 BKK2 RECORD,
      5 BKF2 FLD (CHAR 80),
    4 NAMES2 RECORD,
      5 NM2 FLD (CHAR 132),
    4 BLK2 RECORD,
      5 F2 FLD (CHAR 132),
    4 VALUES2(1:10) RECORD,
```

```
      5 YEAR2 FLD (PIC '9999'),
      5 (TWNX,TWNI,TWNC)
        ARE  FLD(PIC'BB(7)-9V.(6)9'),
    4 BKK3 RECORD,
      5 BKF3 FLD (CHAR 80),
    4 EXD_HD RECORD,
      5 EXD_HDD FLD (CHAR 124),
    4 EXD_B RECORD,
      5 EXD_F FLD (CHAR 124),
    4 NAMES3 RECORD,
      5 NM3 FLD (CHAR 132),
    4 BLK3 RECORD,
      5 F3 FLD (CHAR 132),
    4 VALUES3(1:10) RECORD,
      5 YEAR3 FLD (PIC '9999'),
      5 (OWT$,OPW$,OPM$3,OTWNR,OTWNGP,OXS$3,OMS$3)
        ARE  FLD(PIC'BB(7)-9V.(6)9'):

END_HDD=COPY(' ',42)!!'E N D O G E N O U S   V A R I A B L E S';
EXD_HDD=COPY(' ',42)!!'E X O G E N O U S    V A R I A B L E S';
(END.VALUES1(T),END.VALUES2(T),END.VALUES3(T))=T=SIM_PD;

OWT$=WT$;
OPW$=PW$;
OPM$3=PM$3;
OM$3=M$3;
OPX$3=PX$3;
OX$3=($3;
OTWNPY=TWNPY;
OTWNY=TWNY;
OTWNGP=TWNGP;
OXS$3=XS$3;
OMS$3=MS$3;
OTWNR=CTRTWN.TWNR;

(YEAR1(T),YEAR2(T),YEAR3(T))=START_YR+T;
(F0,F1,F2,F3,BKF1,BKF2,BKF3,EXD_F)=' ';
HFD=COPY(' '.45)!!'LOCAL SIMULATION REPORT FOR: TAIWAN';
NM1='YEAR       M$3       '!!'       PX$3     '
      !!'        X$3      '!!'       TWNPY    '
      !!'        TWNY     '!!'       TWNM     '
      !!'       TWNPX     ';
NM2='YEAR       TWNX      '!!'       TWNI     '
      !!'       TWNC      ';

NM3='YEAR        WT$      '!!'       PW$      '
      !!'        PM$3     '!!'       TWNR     '
      !!'        TWNGP    '!!'       XS$3     '
      !!'        MS$3     ';

T SUBSCRIPT;

BLOCK TWN: MAX ITER IS 40, RELATIVE ERROR IS 0.01;

I1R(T)=DEPENDS_ON(O1R(T));

INITIAL.WT$(T)=IF T<=LAG THEN TS_DATA(4,T)
               ELSE WT$(T-1);
INITIAL.PW$(T)=IF T<=LAG THEN TS_DATA(5,T)
               ELSE PW$(T-1);
INITIAL.PM$3(T)=IF T<=LAG THEN TS_DATA(6,T)
               ELSE PM$3(T-1);

/* THE FOLLOWING EQUATIONS ARE PROVIDED BY MR. YASUDA, LINK PROJECT, 1984 */

BLOCK TAIWANMD: MAX ITER IS 40, RELATIVE ERROR IS 0.01;
```

```
M$3(T)  = IF T>LAG
          THEN  -286.7472 + 0.3144 * TWNY(T)*0.967/40.1000000 +
                0.5442 * M$3(T-1)
                - 519.0132 * PM$3(T)*CTRTWN.TWNR(T)/40.1000000
          ELSE TS_DATA(1,T);

TWNM(T) = IF T>LAG
          THEN (M$3(T) + MS$3(T))* 40.1/0.9611
          ELSE TS_DATA(7,T);

TWNPY(T)= IF T>LAG
          THEN 0.03387 + 0.2889 * TWNY(T)/261558.0 +
               0.6626 * PM$3(T)/1.037*CTRTWN.TWNR(T)/40.1
          ELSE TS_DATA(8,T);

TWNPX(T)= IF T>LAG
          THEN 0.1513 + 0.2915 * TWNPY(T) +
               0.5853 * PW$(T)/1.044*CTRTWN.TWNR(T)/40.1
          ELSE TS_DATA(9,T);

PX$3(T) = IF T>LAG
          THEN 0.05012 + 0.9712 * TWNPX(T)/0.9744/CTRTWN.TWNR(T)*40.1
          ELSE TS_DATA(10,T);

X$3(T)  = IF T>LAG
          THEN -28.1007 + 0.004035 * WT$(T) + 0.8766 * X$3(T-1)
               - 687.1323 * PX$3(T)
          ELSE TS_DATA(2,T);

TWNX(T) = IF T>LAG
          THEN (X$3(T) + XS$3(T))*40.1/0.9744
          ELSE TS_DATA(11,T);

TWNI(T) = IF T>LAG
          THEN -23803.1810 + 0.3534 * TWNY(T)
          ELSE TS_DATA(12,T);

TWNC(T) = IF T>LAG
          THEN 10145.7455 + 0.3091 * TWNY(T) + 0.3876 * TWNC(T-1)
          ELSE TS_DATA(3,T);

TWNY(T) = IF T>LAG
          THEN TWNC(T) + TWNI(T) + TWNGP(T) + TWNX(T) - TWNM(T)
          ELSE TS_DATA(13,T);

END TAIWANMD;
END TWN;
```

A2.6 THE KOREA MODULE

```
MODULE: KOREA;
SOURCE: I1KOREA,TSDKOREA,CTRKRA;
TARGET: O1KOREA,O2KOREA;

1 I1KOREA FILE ORG MAIL,
  2 I1R(*) RECORD,
    3 PROC_ID FLD (CHAR 14),
    3 (WT$,PW$,PM$4)
      ARE FLD (DEC FLOAT(15));

END.I1R(T)=T=SIM_PD;

1 TSDKOREA FILE,
  2 IR(1:18) GRP,
    3 HDR RECORD,
      4 HDD FLD (CHAR 21),
    3 TR(10) RECORD,
      4 TS_DATA FLD (PIC'BB(10)-9V.(7)9');

1 CTRKRA FILE, /* THE CONTROL FILE */
  2 CR RECORD,
    3 START_YR FLD (PIC '9999'), /* STARTING YEAR OF SIMULATION */
    3 LAG      FLD (PIC '9'),    /* LAG FROM THE STARTING YEAR */
    3 SIM_PD   FLD (PIC '99'),   /* SIMULATION PERIOD.         */
  2 I21R RECORD,
    3 HDH FLD (CHAR 109),
  2 I2R(*) RECORD,               /* LOCAL DATABASE             */
    3 HYR FLD (CHAR 4),
    3 (KRAGP,XS$4,MS$4,DUM,KRAR)
      ARE FLD (PIC'BB(10)-9V.(7)9');

1 O1KOREA FILE ORG MAIL,
  2 O1R(*) RECORD,
    3 MAIL_ADR FLD (CHAR 10),
    3 (M$4,PX$4,X$4,KRAPY,KRAY,KRAR)
      ARE FLD (DEC FLOAT(15));

MAIL_ADR='I1KOREAS';

1 O2KOREA FILE,  /* LOCAL RESULTS */
  3 HD RECORD,
    4 HFD FLD (CHAR 132),
  3 BLK0 RECORD,
    4 F0 FLD (CHAR 80),
  3 VALUES GRP,
    4 END_HD RECORD,
      5 END_HDD FLD (CHAR 124),
    4 BKK1 RECORD,
      5 BKF1 FLD (CHAR 80),
    4 NAMES1 RECORD,
      5 NM1 FLD (CHAR 132),
    4 BLK1 RECORD,
      5 F1 FLD (CHAR 132),
    4 VALUES1(1:10) RECORD,
      5 YEAR1 FLD (PIC '9999'),
      5 (QM$4,QPX$4,QX$4,QKRAPY,QKRAY,KRAM,KRAPX)
        ARE  FLD(PIC'BB(7)-9V.(6)9'),
    4 BKK2 RECORD,
      5 BKF2 FLD (CHAR 80),
    4 NAMES2 RECORD,
      5 NM2 FLD (CHAR 132),
    4 BLK2 RECORD,
      5 F2 FLD (CHAR 132),
    4 VALUES2(1:10) RECORD,
```

```
      5 YEAR2 FLD (PIC '9999'),
      5 (KRAX,KRAI,KRAC)
         ARE  FLD(PIC'BB(7)-9V.(6)9'),
    4 BKK3 RECORD,
      5 BKF3 FLD (CHAR 80),
    4 EXD_HD RECORD,
      5 EXD_HDD FLD (CHAR 124),
    4 EXD_B RECORD,
      5 EXD_F FLD (CHAR 124),
    4 NAMES3 RECORD,
      5 NM3 FLD (CHAR 132),
    4 BLK3 RECORD,
      5 F3 FLD (CHAR 132),
    4 VALUES3(1:10) RECORD,
      5 YEAR3 FLD (PIC '9999'),
      5 (OWT$,OPW$,OPM$4,OKRAR,OKRAGP,OXS$4,OMS$4)
         ARE  FLD(PIC'BB(7)-9V.(6)9'),
    4 BKK4 RECORD,
      5 BKF4 FLD (CHAR 80),
    4 NAMES4 RECORD,
      5 NM4 FLD (CHAR 132),
    4 BLK4 RECORD,
      5 F4 FLD (CHAR 132),
    4 VALUES4(1:10) RECORD,
      5 YEAR4 FLD (PIC '9999'),
      5 (ODUM)
         ARE  FLD(PIC'BB(7)-9V.(6)9');

END_HDD=COPY(' ',42)!!'E N D O G E N O U S   V A R I A B L E S';
EXD_HDD=COPY(' ',42)!!'E X O G E N O U S    V A R I A B L E S';
(END.VALUES1(T),END.VALUES2(T),END.VALUES3(T),END.VALUES4(T))=T=SIM_PD;

OWT$=WT$;
OPW$=PW$;
OPM$4=PM$4;
OM$4=M$4;
OPX$4=PX$4;
OX$4=X$4;
OKRAPY=KRAPY;
OKRAY=KRAY;
OKRAGP=KRAGP;
OXS$4=XS$4;
OMS$4=MS$4;
ODUM=DUM;
OKRAR=CTRKRA.KRAR;

(YEAR1(T),YEAR2(T),YEAR3(T),YEAR4(T))=START_YR+T;
(F0,F1,F2,F3,F4,BKF1,BKF2,BKF3,BKF4,EXD_F)=' ';
HFD=COPY(' ',45)!!'LOCAL SIMULATION REPORT FOR: KOREA';
NM1='YEAR       M$4      '!!'       PX$4     '
       !!'        X$4     '!!'       KRAPY    '
       !!'        KRAY    '!!'       KRAM     '
       !!'       KRAPX    ';
NM2='YEAR       KRAX     '!!'       KRAI     '
       !!'       KRAC     ';

NM3='YEAR        WT$     '!!'       PW$      '
       !!'       PM$4     '!!'       KRAR     '
       !!'       KRAGP    '!!'       XS$4     '
       !!'       MS$4     ';
NM4='YEAR        DUM';

T SUBSCRIPT;

BLOCK KRA: MAX ITER IS 20, RELATIVE ERROR IS 0.01;
```

```
I1R(T)=DEPENDS_ON(O1R(T));

INITIAL.WT$(T)=IF T<=LAG THEN TS_DATA(4,T)
               ELSE WT$(T-1);
INITIAL.PW$(T)=IF T<=LAG THEN TS_DATA(5,T)
               ELSE PW$(T-1);
INITIAL.PM$4(T)=IF T<=LAG THEN TS_DATA(6,T)
               ELSE PM$4(T-1);

/* THE FOLLOWING EQUATIONS ARE PROVIDED BY MR. YASUDA, LINK PROJECT, 1984 */

BLOCK KOREAMD: MAX ITER IS 20, RELATIVE ERROR IS 0.01;

 M$4(T)  = IF T>LAG
           THEN -1159.2639 + (0.3239 * KRAY(T)*1000)/316.0000000 +
                (480.3403 * KRAPY(T)/PM$4(T)
                *316.0000000)/CTRKRA.KRAR(T)
           ELSE TS_DATA(1,T);

 KRAM(T) = IF T>LAG
           THEN (M$4(T) + MS$4(T))/1000.0 * 316.0
           ELSE TS_DATA(7,T);

 KRAPY(T)= IF T>LAG
           THEN -0.2485 + 1.0527 * KRAY(T)/2577.36 +
                (0.1906 * PM$4(T)*CTRKRA.KRAR(T))/316.0
           ELSE TS_DATA(8,T);

 KRAPX(T)= IF T>LAG
           THEN 0.2991 + 0.25450 * KRAPY(T) +
               (0.4680 * PW$(T)*CTRKRA.KRAR(T))/316.0
           ELSE TS_DATA(9,T);

 PX$4(T) = IF T>LAG
           THEN -0.07560 + (1.0345 * KRAPX(T)*316.0)/CTRKRA.KRAR(T)
           ELSE TS_DATA(10,T);
     /* *316 IS MOVED TO BEFORE DIVISION */

 X$4(T)  = IF T>LAG
           THEN 462.5595 + 0.004081 * WT$(T) + 0.7485 * X$4(T-1)
                - (1268.9005 * PX$4(T))/PW$(T)
           ELSE TS_DATA(2,T);

 KRAX(T) = IF T>LAG
           THEN (X$4(T) + XS$4(T))/1000.0 * 316.0
           ELSE TS_DATA(11,T);

 KRAI(T) = IF T>LAG
           THEN -225.8597 + 0.3314 * KRAY(T)
           ELSE TS_DATA(12,T);

 KRAC(T) = IF T>LAG
           THEN 38.6775 + 0.1874 * KRAY(T) + 0.7832 * KRAC(T-1) -
                115.1901 * DUM(T)
           ELSE TS_DATA(3,T);

 KRAY(T) = IF T>LAG
           THEN KRAC(T) + KRAI(T) + KRAGP(T) + KRAX(T) - KRAM(T)
           ELSE TS_DATA(13,T);

END KOREAMD;
END KRA;
```

A2.7 THE PHILIPPINE MODULE

```
MODULE: PHILIPPI;
SOURCE: I1PHIL,TSDPHIL, CTRPHIL;
TARGET: O1PHIL,O2PHIL;

1 I1PHIL FILE ORG MAIL,
  2 I1R(*) RECORD,
    3 PROC_ID FLD (CHAR 14),
    3 (WT$,PW$,PM$5)
      ARE FLD (DEC FLOAT(15));

 END.I1R(T)=T=SIM_PD;

1 TSDPHIL FILE,
  2 IR(1:18) GRP,
    3 HDR RECORD,
      4 HDD FLD (CHAR 21),
    3 TR(10) RECORD,
      4 TS_DATA FLD (PIC'BB(10)-9V.(7)9');


1 CTRPHIL FILE, /* THE CONTROL FILE */
  2 CR RECORD,
    3 START_YR FLD (PIC '9999'), /* STARTING YEAR OF SIMULATION */
    3 LAG      FLD (PIC '9'),    /* LAG FROM THE STARTING YEAR */
    3 SIM_PD   FLD (PIC '99'),   /* SIMULATION PERIOD.         */
  2 I21R record,
    3 DHD FLD (CHAR 88),
  2 I2R(*) RECORD,               /* LOCAL DATABASE             */
    3 HYR FLD (CHAR 4),
    3 (PHIGP,XS$5,MS$5,PHIR)
      ARE FLD (PIC'BB(10)-9V.(7)9');

1 O1PHIL FILE ORG MAIL,
  2 O1R(*) RECORD,
    3 MAIL_ADR FLD (CHAR 10),
    3 (M$5,PX$5,X$5,PHIPY,PHIY,PHIR)
      ARE FLD (DEC FLOAT(15));

MAIL_ADR='I1PHILS';

1 O2PHIL FILE,  /* LOCAL RESULTS */
  3 HD RECORD,
    4 HFD FLD (CHAR 132),
  3 BLK0 RECORD,
    4 F0 FLD (CHAR 80),
  3 VALUES GRP,
    4 END_HD RECORD,
      5 END_HDD FLD (CHAR 124),
    4 BKK1 RECORD,
      5 BKF1 FLD (CHAR 80),
    4 NAMES1 RECORD,
      5 NM1 FLD (CHAR 132),
    4 BLK1 RECORD,
      5 F1 FLD (CHAR 132),
    4 VALUES1(1:10) RECORD,
      5 YEAR1 FLD (PIC '9999'),
      5 (0M$5,0PX$5,0X$5,0PHIPY,0PHIY,PHIM,PHIPX)
        ARE  FLD(PIC'BB(7)-9V.(6)9'),
    4 BKK2 RECORD,
      5 BKF2 FLD (CHAR 80),
    4 NAMES2 RECORD,
      5 NM2 FLD (CHAR 132),
    4 BLK2 RECORD,
      5 F2 FLD (CHAR 132),
```

```
    4 VALUES2(1:10) RECORD,
      5 YEAR2 FLD (PIC '9999'),
      5 (PHIX,PHII,PHIC)
         ARE  FLD(PIC'BB(7)-9V.(6)9'),
    4 BKK3 RECORD,
      5 BKF3 FLD (CHAR 80),
    4 EXD_HD RECORD,
      5 EXD_HDD FLD (CHAR 124),
    4 EXD_B RECORD,
      5 EXD_F FLD (CHAR 124),
    4 NAMES3 RECORD,
      5 NM3 FLD (CHAR 132),
    4 BLK3 RECORD,
      5 F3 FLD (CHAR 132),
    4 VALUES3(1:10) RECORD,
      5 YEAR3 FLD (PIC '9999'),
      5 (OWT$,OPW$,OPM$5,OPHIR,OPHIGP,OXS$5,OMS$5)
         ARE  FLD(PIC'BB(7)-9V.(6)9');

END_HDD=COPY(' ',42)!!'E N D O G E N O U S   V A R I A B L E S';
EXD_HDD=COPY(' ',42)!!'E X O G E N O U S    V A R I A B L E S';
(END.VALUES1(T),END.VALUES2(T),END.VALUES3(T))=T=SIM_PD;

OWT$=WT$;
OPW$=PW$;
OPM$5=PM$5;
OM$5=M$5;
OPX$5=PX$5;
OX$5=X$5;
OPHIPY=PHIPY;
OPHIY=PHIY;
OPHIGP=PHIGP;
OXS$5=XS$5;
OMS$5=MS$5;
OPHIR=CTRPHIL.PHIR;


(YEAR1(T),YEAR2(T),YEAR3(T))=START_YR+T;
(F0,F1,F2,F3,BKF1,BKF2,BKF3,EXD_F)=' ';
HFD=COPY(' ',45)!!'LOCAL SIMULATION REPORT FOR: PHILIPPINE';
NM1='YEAR      M$5      '!!'      PX$5     '
      !!'       X$5     '!!'      PHIPY    '
      !!'       PHIY    '!!'      PHIM     '
      !!'      PHIPX    ';
NM2='YEAR      PHIX     '!!'      PHII     '
      !!'      PHIC     ';

NM3='YEAR       WT$     '!!'      PW$      '
      !!'       PM$5    '!!'      PHIR     '
      !!'       PHIGP   '!!'      XS$5     '
      !!'       MS$5    ';

T SUBSCRIPT;

BLOCK PHI: MAX ITER IS 20, RELATIVE ERROR IS 0.01;

I1R(T)=DEPENDS_ON(O1R(T));

INITIAL.WT$(T)=IF T<=LAG THEN TS_DATA(4,T)
               ELSE WT$(T-1);
INITIAL.PW$(T)=IF T<=LAG THEN TS_DATA(5,T)
               ELSE PW$(T-1);
INITIAL.PM$5(T)=IF T<=LAG THEN TS_DATA(6,T)
               ELSE PM$5(T-1);

/* THE FOLLOWING EQUATIONS ARE PROVIDED BY MR. YASUDA, LINK PROJECT, 1984 */
```

```
BLOCK PHILMD: MAX ITER IS 20, RELATIVE ERROR IS 0.01;

 M$5(T)  = IF TDLAG
           THEN  -491.7211 + 0.1231 * PHIY(T)*1.257
                 /5.7290 + 0.5040 * M$5(T-1)
                 + 258.14210 * PHIPY(T)/1.2570
                 /PM$5(T)*5.7290/CTRPHIL.PHIR(T)
           ELSE TS_DATA(1,T);

 PHIM(T) = IF TDLAG
           THEN (M$5(T) + MS$5(T))*5.729/1.586
           ELSE TS_DATA(7,T);

 PHIPY(T)= IF TDLAG
           THEN 0.1296 + 0.3945 * PHIY(T)/29515.0 +
                0.4552 * PM$5(T)/0.9140 *CTRPHIL.PHIR(T)/3.9
           ELSE TS_DATA(8,T);

 PHIPX(T)= IF TDLAG
           THEN -0.75110 + 1.8239 * PHIPY(T)
           ELSE TS_DATA(9,T);

 PX$5(T) = IF TDLAG
           THEN -0.02338 + 1.0404 * PHIPX(T)/1.757/CTRPHIL.PHIR(T)*5.7290
           ELSE TS_DATA(10,T);

 X$5(T)  = IF TDLAG
           THEN 1006.1559 + 0.001548 * WT$(T) + 0.3555 * X$5(T-1)
                - 816.6783000 * PX$5(T)/PW$(T)
           ELSE TS_DATA(2,T);

 PHIX(T) = IF TDLAG
           THEN (X$5(T) + XS$5(T))*5.729/1.757
           ELSE TS_DATA(11,T);

 PHII(T) = IF TDLAG
           THEN -3244.7515 + 0.3101 * PHIY(T)
           ELSE TS_DATA(12,T);

 PHIC(T) = IF TDLAG
           THEN 692.4862 + 0.08436 * PHIY(T) + 0.8906 * PHIC(T-1)
           ELSE TS_DATA(3,T);

 PHIY(T) = IF TDLAG
           THEN PHIC(T) + PHII(T) + PHIGP(T) + PHIX(T) - PHIM(T)
           ELSE TS_DATA(13,T);

END PHILMD;
END PHI;
```

A2.8 THE THAILAND MODULE

```
MODULE: THAILAND;
SOURCE: I1THAI,TSDTHAI, CTRTHI;
TARGET: O1THAI,O2THAI;

1 I1THAI FILE ORG MAIL,
  2 I1R(*) RECORD,
    3 PROC_ID FLD (CHAR 14),
    3 (WT$,PW$,PM$6)
       ARE FLD (DEC FLOAT(15));

 END.I1R(T)=T=SIM_PD;

1 TSDTHAI FILE,
  2 IR(1:18) GRP,
    3 HDR RECORD,
      4 HDD FLD (CHAR 21),
    3 TR(10) RECORD,
      4 TS_DATA FLD (PIC'BB(10)-9V.(7)9');

1 CTRTHI FILE, /* THE CONTROL FILE */
  2 CR RECORD,
    3 START_YR FLD (PIC '9999'), /* STARTING YEAR OF SIMULATION */
    3 LAG      FLD (PIC '9'),    /* LAG FROM THE STARTING YEAR */
    3 SIM_PD   FLD (PIC '99'),   /* SIMULATION PERIOD.         */
  2 I21r record,
    3 DHD FLD (CHAR 88),
  2 I2R(*) RECORD,
    3 HYR FLD (CHAR 4),
    3 (THIGP,XS$6,MS$6,THIR)
       ARE FLD (PIC'BB(10)-9V.(7)9');

1 O1THAI FILE ORG MAIL, /* SEND TO THE WORLD MODULE */
  2 O1R(*) RECORD,
    3 MAIL_ADR FLD (CHAR 10),
    3 (M$6,PX$6,X$6,THIPY,THIY,THIR)
       ARE FLD (DEC FLOAT(15));

MAIL_ADR='I1THAIS';

1 O2THAI FILE,  /* LOCAL RESULTS */
  3 HD RECORD,
    4 HFD FLD (CHAR 132),
  3 BLK0 RECORD,
    4 F0 FLD (CHAR 80),
  3 VALUES GRP,
    4 END_HD RECORD,
      5 END_HDD FLD (CHAR 124),
    4 BKK1 RECORD,
      5 BKF1 FLD (CHAR 80),
    4 NAMES1 RECORD,
      5 NM1 FLD (CHAR 132),
    4 BLK1 RECORD,
      5 F1 FLD (CHAR 132),
    4 VALUES1(1:10) RECORD,
      5 YEAR1 FLD (PIC '9999'),
      5 (OM$6,OPX$6,OX$6,OTHIPY,OTHIY,THIM,THIPX)
         ARE  FLD(PIC'BB(7)-9V.(6)9'),
    4 BKK2 RECORD,
      5 BKF2 FLD (CHAR 80),
    4 NAMES2 RECORD,
      5 NM2 FLD (CHAR 132),
    4 BLK2 RECORD,
      5 F2 FLD (CHAR 132),
    4 VALUES2(1:10) RECORD,
```

```
      5 YEAR2 FLD (PIC '9999'),
      5 (THIX,THII,THIC)
        ARE  FLD(PIC'BB(7)-9V.(6)9'),
    4 BKK3 RECORD,
      5 BKF3 FLD (CHAR 80),
    4 EXD_HD RECORD,
      5 EXD_HDD FLD (CHAR 124),
    4 EXD_B RECORD,
      5 EXD_F FLD (CHAR 124),
    4 NAMES3 RECORD,
      5 NM3 FLD (CHAR 132),
    4 BLK3 RECORD,
      5 F3 FLD (CHAR 132),
    4 VALUES3(1:10) RECORD,
      5 YEAR3 FLD (PIC '9999'),
      5 (OWT$,OPW$,OPM$6,OTHIR,OTHIGP,OXS$6,OMS$6)
        ARE  FLD(PIC'BB(7)-9V.(6)9');

END_HDD=COPY(' ',42)!!'E N D O G E N O U S   V A R I A B L E S';
EXD_HDD=COPY(' ',42)!!'E X O G E N O U S    V A R I A B L E S';
(END.VALUES1(T),END.VALUES2(T),END.VALUES3(T))=T=SIM_PD;

OWT$=WT$;
OPW$=PW$;
OPM$6=PM$6;
OM$6=M$6;
OPX$6=PX$6;
OX$6=X$6;
OTHIPY=THIPY;
OTHIY=THIY;
OTHIGP=THIGP;
OXS$6=XS$6;
OMS$6=MS$6;
OTHIR=CTRTHI.THIR;

(YEAR1(T),YEAR2(T),YEAR3(T))=START_YR+T;
(F0,F1,F2,F3,BKF1,BKF2,BKF3,EXD_F)=' ';
HFD=COPY(' ',45)!!'LOCAL SIMULATION REPORT FOR: THAILAND';
NM1='YEAR       M$6      '!!'       PX$6     '
      !!'        X$6     '!!'       THIPY    '
      !!'        THIY    '!!'       THIM     '
      !!'       THIPX    ';
NM2='YEAR       THIX     '!!'       THII     '
      !!'       THIC     ';

NM3='YEAR        WT$     '!!'       PW$      '
      !!'        PM$6    '!!'       THIR     '
      !!'        THIGP   '!!'       XS$6     '
      !!'        MS$6    ';

T SUBSCRIPT;

BLOCK THI: MAX ITER IS 40, RELATIVE ERROR IS 0.01;

IIR(T)=DEPENDS_ON(OIIR(T));

INITIAL.WT$(T)=IF T<=LAG THEN TS_DATA(4,T)
               ELSE WT$(T-1);
INITIAL.PW$(T)=IF T<=LAG THEN TS_DATA(5,T)
               ELSE PW$(T-1);
INITIAL.PM$6(T)=IF T<=LAG THEN TS_DATA(6,T)
               ELSE PM$6(T-1);

/* THE FOLLOWING EQUATIONS ARE PROVIDED BY MR. YASUDA, LINK PROJECT, 1984 */

BLOCK THAIMD: MAX ITER IS 40, RELATIVE ERROR IS 0.01;
```

```
MS$6(T)  = IF T>LAG
           THEN -820.4688 + 0.1252 * THIY(T)*1.1      - 0.3850 * MS$6(T-1)
                + 763.0320 * THIPY(T)/1.135
                /PM$6(T)/CTRTHI.THIR(T)*20.930
           ELSE TS_DATA(1,T);

THIM(T)  = IF T>LAG
           THEN (M$6(T) + MS$6(T))*20.93/1.08
           ELSE TS_DATA(7,T);

THIPY(T)= IF T>LAG
           THEN 0.4292 + 0.07972 * THIY(T)/63793.0 +
                0.4995 * PM$6(T)/0.87*CTRTHI.THIR(T)/20.84
           ELSE TS_DATA(8,T);

THIPX(T)= IF T>LAG
           THEN -0.2305 + 0.5310 * THIPY(T) +
                0.6188 * PW$(T)/0.973*CTRTHI.THIR(T)/20.84
           ELSE TS_DATA(9,T);

PX$6(T) = IF T>LAG
           THEN -0.06976 + 1.0885 * THIPX(T)/1.016/CTRTHI.THIR(T)*20.93
           ELSE TS_DATA(10,T);

X$6(T)   = IF T>LAG
           THEN 567.0480 + 0.0009513 * WT$(T) + 0.5164 * X$6(T-1)
                - 416.5699 * PX$6(T)/PW$(T)
           ELSE TS_DATA(2,T);

THIX(T) = IF T>LAG
           THEN (X$6(T) + XS$6(T))*20.93/1.016
           ELSE TS_DATA(11,T);

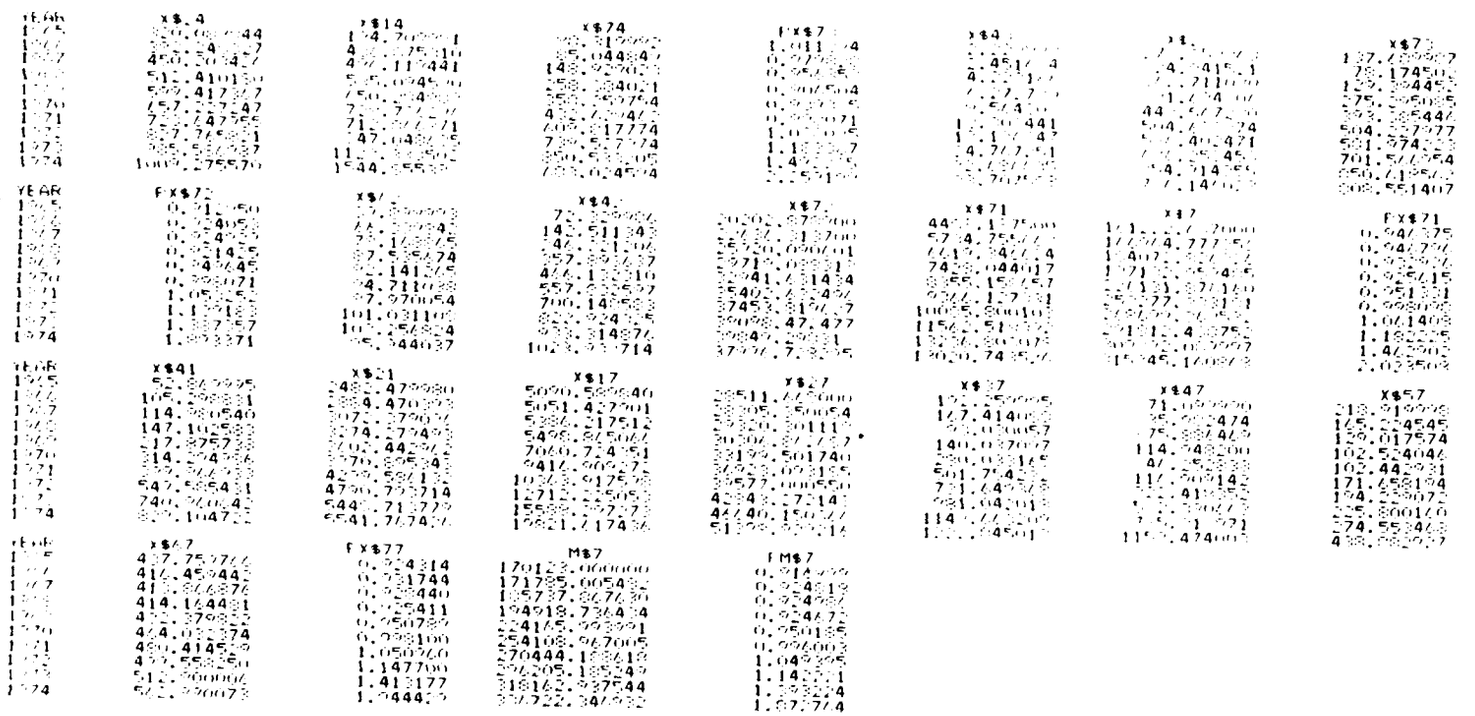
THII(T) = IF T>LAG
           THEN -3871.7198 + 0.2747 * THIY(T)
           ELSE TS_DATA(12,T);

THIC(T) = IF T>LAG
           THEN 2903.0806 + 0.2609 * THIY(T) + 0.6197 * THIC(T-1)
           ELSE TS_DATA(3,T);

THIY(T) = IF T>LAG
           THEN THIC(T) + THII(T) + THIGP(T) + THIX(T) - THIM(T)
           ELSE TS_DATA(13,T);

END THAIMD;
END THI;
```

A2.9 SIMULATION RESULTS



EXOGENOUS VARIABLES

YEAR	M$2	FX$1	X$1	IBAF	IBAY	IBAR	M$	FX$3
1975	3571.007800	0.955000	3121.477000	0.741000	1.000000	1.000000	777.300000	0.957000
1976	3771.100000	0.911887	3445.116540	0.801000	1.000000	1.000000	750.120000	0.914000
1977	3981.125591	0.904816	3775.435595	0.861000	1.000000	1.000000	777.240000	0.925000
1978	4201.144138	0.915774	4060.000000	0.921000	1.000000	1.000000	811.440000	0.926067
1979	4052.433000	0.955774	4435.390641	0.980000	1.000000	1.000000	844.800000	0.957000
1980	4784.000654	0.977455	4261.811203	0.980000	1.000000	1.000000	877.740000	0.957000
1981	4987.842575	1.013040	4577.436715	0.940415	1.000000	1.000000	911.280000	0.957000
1982	4844.962744	1.067416	4977.436715	0.978043	1.000000	1.000000	945.420000	0.957000
1983	4957.000000	1.120550	5454.015307	1.017572	1.000000	1.000000	980.160000	0.957000
1984	4755.683066	1.160751	5955.455000	1.148000	1.000000	1.000000	1015.500000	0.957000

YEAR	FX$1	X$1	IAFY	IAFY	IAFR	M$	FX$3
1975	0.955000	3010.477000	0.781000	4007.400000	3010.477000	777.300000	0.957000
1976	0.910500	10287.351177	0.820550	4772.170511	3441.477754	750.120000	0.914000
1977	0.904524	11770.887431	0.859007	5081.795527	3441.477754	777.240000	0.925000
1978	0.947415	12485.766522	0.876004	5421.151222	3441.477754	811.440000	0.926067
1979	0.975751	15574.085724	0.927314	6042.710000	3441.477754	844.800000	0.957000
1980	1.017043	18724.425379	0.987774	6456.234000	3441.477754	877.740000	0.957000
1981	1.122447	20085.724044	0.999673	7145.942705	3441.477754	911.280000	1.029711
1982	1.155000	22457.115143	1.060414	7851.555000	3441.477754	945.420000	1.073700
1983	1.444511	20007.483541	1.140513	8764.340000	3441.477754	980.160000	1.154400
1984	1.441072	20007.483541	1.140513	8764.340000	3441.477754	1015.500000	1.170000

YEAR	X$1	TWNFY	TWNY	TWNR	M$4	FX$4	X$4
1975	4134.700041	0.754000	150096.000000	40.000000	4231.247754	0.955000	4205.210004
1976	5000.000041	0.790100	162548.000000	40.000000	4943.040500	0.910500	5119.875254
1977	5767.125000	0.812000	173000.000000	40.000000	5128.000471	0.904524	5755.960457
1978	6511.400001	0.840400	184955.000000	40.000000	5345.647447	0.947415	4037.771751
1979	7104.666666	0.881050	214218.650000	40.000000	5454.445217	0.975751	4564.304464
1980	7600.075000	0.901140	235867.750000	40.000000	5674.841444	0.971000	1013.407555
1981	8000.000000	1.004154	254770.155734	40.000000	5110.000000	0.945000	1454.362400
1982	8415.666666	1.026750	265116.407716	40.000000	5430.650204	1.000000	1743.875520
1983	8744.400000	1.447004	340787.175542	40.000000	5053.111732	1.177111	2457.771007
1984	8969.143200	1.472118	303779.522915	40.000000	5354.950075	1.000000	2065.710000

YEAR	IBAFY	IBAY	IBAR	M$5	FX$5	X$5	FNILFY
1975	0.554000	1500.250770	271.000000	1062.017750	0.827000	1.111827	0.000000
1976	0.577711	1476.577847	270.000000	1113.130414	0.850187	1.220000	0.924400
1977	0.668000	1384.124771	274.000000	1188.550507	0.884771	1.265427	0.955400
1978	0.727402	1327.172000	281.000000	1243.264774	0.901665	1.141140	0.994000
1979	0.748407	1307.716662	304.000000	1314.947402	0.957700	1.144140	1.020000
1980	0.965615	1508.900000	314.000000	1334.111718	0.940205	1.277794	1.304000
1981	1.132447	1774.400400	323.000000	1392.397062	1.000427	1.111827	1.464511
1982	1.145000	2146.000000	399.000000	1457.114275	1.007540	1.157417	1.613015
1983	1.444511	1750.000000	399.000000	1507.187457	1.300734	1.700000	1.864000
1984	1.441072	4054.000000	399.000000	1551.111000	1.441552	1.700000	2.175000

YEAR	FHIY	FHIR	M$/	FX$/	X$/	THIFY	THIY
1965	27135.000000	1.000000	709.254744	1.014999	410.442971	1.000000	7948?.000000
1966	27948.455541	3.000000	847.018847	1.000199	485.112737	1.071144	85249.355022
1967	28550.845111	3.000000	975.000207	1.007994	765.514709	1.084144	90330.757511
1968	30107.784143	3.000000	1093.074408	1.014447	485.117772	1.094154	100759.952277
1969	31742.179957	3.000000	1177.746363	1.040779	739.501218	1.161122	115914.940394
1970	33183.842079	5.750000	1599.454013	1.104144	734.224462	1.154641	122125.107121
1971	35384.852689	6.204999	1490.127380	1.171744	727.444456	1.211754	125271.472200
1972	37114.915640	6.681999	1332.034407	1.271214	872.172805	1.275527	135773.674[illegible]
1973	38400.982704	6.774999	1324.255541	1.647541	1.555[illegible]	1.[illegible]4[illegible]	137776.76[illegible]69
1974	39632.092751	6.799999	1564.206032	2.071[illegible]	1.[illegible]4[illegible]	1.747099	153620.184944

YEAR	THIR
1965	20.829994
1966	20.749994
1967	20.799987
1968	20.849979
1969	20.929993
1970	20.959975
1971	20.959975
1972	20.929993
1973	20.879989
1974	20.879989

FINAL SIMULATION RESULTS FOR USA

ENDOGENOUS VARIABLES

YEAR	M$	EX$	X$	USAFY	USAY	USAM	USAFX
1965	2571.007800	0.155000	4121.476600	0.743100	925.000000	41.000000	0.804000
1966	2691.727171	0.891198	4480.155800	0.801443	965.343071	47.049874	0.835962
1967	3183.036292	0.704835	5010.710657	0.850856	1027.771613	51.684349	0.884640
1968	3631.483115	0.915294	5113.418145	0.847445	1073.804572	60.542576	0.858457
1969	4052.472477	0.939803	5500.579714	0.885936	1101.842440	67.176841	0.881150
1970	4276.179443	0.777487	4270?.112258	0.909707	1123.767545	71.105938	0.916185
1971	4284.976539	1.019236	4567?.904561	0.940551	1151.327650	70.842179	0.954999
1972	4344?.780446	1.057509	4984?.020543	0.979776	1192.934?.1	75.650014	1.018472
1973	4956?.990463	1.270796	5460?.804369	1.063660	1224.848?.?	80.021755	1.188781
1974	4792?.431135	1.203791	5?988.042869	1.168310	1208.082486	74.500777	1.496450

YEAR	USAX	USAI	USA
1965	49.000000	150.000991	558.000854
1966	51.374486	147.738233	583.979103
1967	64.080795	158.522355	615.544659
1968	58.202401	168.116451	648.849645
1969	62.020974	173.388147	676.930409
1970	67.043923	177.502836	700.124731
1971	68.238650	182.678678	721.875627
1972	73.017?71	188.418356	743.849899
1973	87.087?03	194.496078	768.784241
1974	92.244005	142.160217	770.059996

EXOGENOUS VARIABLES

YEAR	WT$	PW$	PM$?	USAR	USAGF	X*$?	MS$?
1965	204586.437000	0.913999	0.887999	1.000000	207.599991	424999.375000	12959.972700
1966	211748.797731	0.923435	0.922558	1.000000	229.?99988	445816.312000	14929.710900
1967	231474.533718	0.924995	0.924016	1.000000	248.?99988	479481.500000	16449.718700
1968	247133.421300	0.924320	0.926828	1.000000	259.199951	506750.937000	17641.304700
1969	264?64.084544	0.950435	0.954269	1.000000	264.699951	538914.875000	19326.093700
1970	275299.427712	0.998071	1.000604	1.000000	250.199997	581469.875000	19549.205100
1971	288286.996081	1.047908	1.043642	1.000000	249.399994	589421.312000	19755.894500
1972	368669.421114	1.136716	1.149061	1.000000	253.099991	629964.500000	18963.144100
1973	395437.713436	1.388926	1.390328	1.000000	252.499985	756202.312000	21943.367500
1974	412845.009869	1.878756	1.843149	1.000000	257.699951	803814.687000	19057.761700

YEAR	DUM
1965	0.000000
1966	0.000000
1967	0.000000
1968	0.000000
1969	0.000000
1970	0.000000
1971	0.000000
1972	0.000000
1973	0.000000
1974	1.000000

$

GLOBAL SIMULATION REPORT FOR: JAPAN

ENDOGENOUS VARIABLES

YEAR	M$1	PX$1	X$1	JAPPY	JAPY	JAPM	JAPPX
1965	7822.476560	0.925999	9010.677900	0.781999	40874.476100	514.499700	0.887999
1966	9416.268748	0.930947	10300.397085	0.820744	44740.710200	4105.042990	0.916331
1967	10785.990595	0.932376	11798.750936	0.850000	50658.504277	4735.573740	0.925414
1968	11892.218201	0.947740	12988.470495	0.877163	54963.580159	5426.475299	0.932547
1969	13338.508384	0.975905	15422.412135	0.927851	60524.455271	6108.158470	0.955922
1970	14953.560005	1.017308	18793.584259	0.988005	66648.496985	7088.262493	0.990455
1971	16293.305778	1.138895	20172.764185	1.000541	71540.595349	7938.995573	0.966963
1972	18128.190653	1.225537	22538.064996	1.069782	78600.190259	8553.339143	1.003989
1973	20560.808104	1.444927	25689.065422	1.182371	87928.024116	9169.973944	1.091590
1974	20518.819752	1.661273	30024.835599	1.340793	96677.301410	10468.021950	1.243 [illegible]

YEAR	JAPX	JAPI	JAPY
1965	4012.000000	12382.496100	23448.097700
1966	4475.033504	16014.139087	25589.182779
1967	5037.569480	17868.498867	27688.212600
1968	5613.277415	19752.045073	29766.334299
1969	6712.780337	22338.268288	32177.365994
1970	8044.096495	25001.797326	34895.467997
1971	8571.911397	27210.198893	37484.180632
1972	9555.059987	30410.593089	40524.507884
1973	10927.189830	34639.428555	44343.731015
1974	13344.567460	39431.528168	48655.329295

EXOGENOUS VARIABLES

YEAR	WT$	PW$	PM$1	JAPR	JAPGR	X$1	M$1
1965	204586.437000	0.912999	0.904999	340.899902	4549.996870	9288.564210	9005.546870
1966	211748.793731	0.923425	0.926259	362.499756	4767.996440	11117.677710	10633.157960
1967	231474.532718	0.934996	0.924976	341.899902	4998.796870	9398.412600	9456.665200
1968	247133.421300	0.924320	0.924663	357.699951	5283.396440	9708.614010	9887.480110
1969	285549.084544	0.950435	0.950508	357.799805	5504.196120	9149.344970	9374.147120
1970	317529.627712	0.998071	0.992964	357.599854	5796.096440	9701.092770	9868.303150
1971	338286.896051	1.047908	1.047998	314.799805	6211.000000	9797.905270	9907.446840
1972	368569.421116	1.136716	1.150160	302.000000	6696.896440	10899.535640	10790.542970
1973	395687.713636	1.288934	1.396992	280.000000	7187.596760	12417.953120	12085.592970
1974	412845.009859	1.373736	1.577995	301.000000	7502.896440	13937.197500	13256.452190

YEAR	DUM
1965	0.000000
1966	0.000000
1967	0.000000
1968	0.000000
1969	0.000000
1970	0.000000
1971	0.000000
1972	0.000000
1973	0.000000
1974	1.000000

$

LOCAL SIMULATION REPORT FOR: KOREA

E N D O G E N O U S V A R I A B L E S

YEAR	M$4	PX$4	X$4	KRAPY	KRAY	KRAM	KRAPX
1965	631.249756	0.852999	205.219986	0.524999	1520.759770	149.549988	0.854999
1966	943.466060	0.916604	220.786462	0.593886	1696.892658	244.665395	0.819497
1967	1129.814539	0.929283	297.683336	0.659349	1848.993275	313.052662	0.842265
1968	1346.881386	0.938039	406.192650	0.736897	2028.102059	394.728271	0.871309
1969	1658.093780	0.944063	659.283912	0.869217	2307.877079	466.108389	0.948228
1970	1877.059517	0.971486	1016.770314	0.966482	2510.525587	621.681823	1.012167
1971	2112.814214	0.945400	1459.385244	1.127679	2778.495514	779.751063	1.164977
1972	2493.400446	0.994472	1948.926245	1.317341	3147.346979	914.614108	1.306076
1973	3055.323026	1.176371	2461.418866	1.597131	3691.753920	1080.850312	1.524263
1974	3337.895985	1.368632	3065.376398	1.935106	4059.538507	1133.997714	2.138281

YEAR	KRAX	KRAI	KRAC
1965	80.289993	197.059998	1201.119870
1966	107.540662	336.384879	1247.332524
1967	156.166673	386.674970	1401.124304
1968	220.336793	445.944049	1515.929493
1969	312.997221	538.586165	1658.232160
1970	437.234773	605.589996	1807.572887
1971	577.333308	694.308352	1974.705015
1972	745.179141	816.520656	2174.711485
1973	1005.391237	997.127225	2433.485916
1974	1097.158536	1119.238446	2590.029376

E X O G E N O U S V A R I A B L E S

YEAR	WT$	PW$	PM$4	KRAR	KRAGP	XS$4	MS$4
1965	204586.437000	0.912999	0.888999	271.000000	181.559998	48.862258	-157.990295
1966	211748.793731	0.923425	0.917603	270.000000	200.299988	119.532089	-169.208481
1967	231474.532718	0.924995	0.925275	274.000000	218.079987	196.514999	-139.141556
1968	247133.421300	0.924320	0.928781	281.000000	240.619995	291.075684	-97.741287
1969	282549.084544	0.950435	0.957444	304.000000	264.169922	331.213623	-183.067230
1970	317529.627712	0.998071	0.998134	316.000000	281.809814	366.884033	90.288024
1971	338286.896081	1.047908	1.076084	373.000000	311.899902	367.618896	354.752441
1972	368569.421116	1.136716	1.168179	399.000000	325.549805	409.235596	400.947998
1973	395687.713636	1.388926	1.409375	398.000000	336.599854	720.198975	365.089355
1974	412845.009859	1.878736	1.801143	484.000000	387.109863	406.644287	250.704376

YEAR	DUM
1965	0.000000
1966	0.000000
1967	0.000000
1968	0.000000
1969	0.000000
1970	0.000000
1971	0.000000
1972	0.000000
1973	0.000000
1974	1.000000

$

LOCAL SIMULATION REPORT FOR: TAIWAN

ENDOGENOUS VARIABLES

YEAR	M$3	PX$3	X$3	TWNPY	TWNY	TWNM	TWNPX
1965	667.309814	0.927999	484.709961	0.754699	150696.000000	28680.000000	0.831699
1966	750.129883	0.916999	582.939941	0.780199	162348.000000	29895.000000	0.877799
1967	939.249756	0.962999	657.129883	0.816099	179003.000000	37772.000000	0.909199
1968	1242.239990	0.962999	820.349854	0.878399	195106.000000	49304.000000	0.925499
1969	1417.829830	0.962999	1077.579830	0.925499	211588.000000	57140.000000	0.944399
1970	1695.969970	1.000000	1424.519780	0.966999	234573.000000	69782.000000	0.974399
1971	1938.129880	1.023999	1946.879880	1.000000	261558.000000	85261.000000	1.000000
1972	2266.509770	1.118999	2604.809810	1.050099	292625.000000	100918.000000	1.079599
1973	3174.769780	1.389999	3148.639890	1.186199	327698.000000	124433.000000	1.258999
1974	2930.543594	1.716787	3240.099283	1.495629	315947.267913	143278.750267	1.586674

YEAR	TWNX	TWNI	TWNC
1965	25016.000000	28796.000000	90460.000000
1966	29693.000000	32221.000000	93842.000000
1967	34652.000000	40082.000000	102671.000000
1968	43619.000000	47626.000000	111157.000000
1969	54575.000000	49879.000000	119953.000000
1970	68713.000000	60186.000000	129964.000000
1971	91747.000000	67565.000000	141096.000000
1972	121611.000000	68650.000000	154814.000000
1973	150582.000000	81581.000000	170898.000000
1974	148241.352848	87772.019504	173974.645828

EXOGENOUS VARIABLES

YEAR	WT$	PW$	PM$3	TWNR	TWNGP	XS$3	MS$3
1965	204586.437000	0.912999	0.894999	40.099990	35104.000000	123.159988	20.080246
1966	221354.437000	0.920999	0.911999	40.099990	36487.000000	138.577576	-33.619201
1967	232835.562000	0.923999	0.917999	40.099990	39370.000000	184.887604	-33.946487
1968	259325.000000	0.922999	0.926999	40.099990	42008.000000	239.558975	-60.542633
1969	287506.000000	0.949999	0.952999	40.099990	44321.000000	248.551544	-48.322555
1970	312508.500000	1.000000	1.000000	40.099990	45492.000000	245.154343	-23.464599
1971	333437.750000	1.043999	1.036999	40.099990	46411.000000	282.502930	105.369583
1972	367282.187000	1.127999	1.130999	40.099990	48467.000000	350.245850	152.250031
1973	413530.500000	1.382999	1.396999	38.099990	49070.000000	510.389160	-192.412430
1974	418280.636087	1.878671	1.835804	38.049987	49238.000000	362.054687	503.501465

$

LOCAL SIMULATION REPORT FOR: PHILIPPINE

ENDOGENOUS VARIABLES

YEAR	M$5	PX$5	X$5	PHIPY	PHIY	PHIM	PHIPX
1965	1062.019780	0.827999	931.119873	0.898999	27135.000000	4068.000000	0.904999
1966	1118.760194	0.840057	934.921606	0.956045	28036.055852	4370.139289	0.992631
1967	1195.576056	0.878997	922.046866	0.980590	29611.541684	4678.331055	1.037398
1968	1252.504642	0.896804	941.877589	0.991814	30227.759064	4938.486768	1.057870
1969	1326.451537	0.953131	966.682378	1.027317	31840.635256	5028.090454	1.122624
1970	1341.709875	0.941319	1064.819793	1.305037	33206.684612	5114.002747	1.629158
1971	1399.438923	0.989275	1126.991429	1.443711	35428.133408	5000.217386	1.882085
1972	1464.527262	1.075265	1196.855361	1.598276	37167.022750	6108.464416	2.163997
1973	1517.032871	1.305070	1301.122896	1.866418	38545.331220	6773.323655	2.653061
1974	1558.250145	1.664754	1392.516840	2.264626	39743.080670	7034.985555	3.379352

YEAR	PHIX	PHII	PHIC
1965	4264.000000	5184.000000	19319.000000
1966	4254.857138	5449.228879	20263.109124
1967	4573.243938	5937.787877	21236.840924
1968	4233.339336	6128.876137	22156.030359
1969	4226.972762	6629.029986	23110.722962
1970	4508.830476	7052.644151	24076.212732
1971	4658.354896	7741.516314	25123.479584
1972	5620.872085	8280.746712	26202.868369
1973	5917.029379	8708.175266	27280.450230
1974	5551.636483	9077.746481	28340.683261

EXOGENOUS VARIABLES

YEAR	WT$	PW$	PM$5	PHIR	PHIGP	XS$5	MS$5
1965	204586.437000	0.912999	0.895999	3.899999	2436.000000	376.585449	64.153533
1966	221354.437000	0.920999	0.906999	3.899999	2439.000000	369.980469	91.056686
1967	232835.562000	0.923999	0.913999	3.899999	2542.000000	480.499756	99.559753
1968	259325.000000	0.922999	0.919999	3.899999	2648.000000	356.425293	114.651932
1969	287506.000000	0.949999	0.947999	3.899999	2902.000000	329.667969	65.510665
1970	312508.500000	1.000000	1.000000	5.728999	2683.000000	317.972168	74.036041
1971	333437.750000	1.043999	1.043999	6.304999	2905.000000	301.657471	-15.193020
1972	367282.187000	1.127999	1.138999	6.681999	3171.000000	526.983398	226.522583
1973	413530.500000	1.382999	1.411999	6.774999	3418.000000	512.009521	358.074707
1974	418280.636087	1.878671	1.849486	6.790999	3808.000000	310.038379	389.295166

$

LOCAL SIMULATION REPORT FOR: THAILAND

E N D O G E N O U S V A R I A B L E S

YEAR	M$6	PX$6	X$6	THIPY	THIY	THIM	THIPX
1965	709.259766	1.014999	610.469971	1.059999	79487.000000	15884.000000	1.000000
1966	868.256342	0.999682	640.711029	1.073677	92580.807714	18390.386602	0.989626
1967	983.178193	1.006156	665.798682	1.082531	99563.850119	21803.124833	0.998016
1968	1099.900572	1.013512	700.142290	1.098306	109330.892251	23887.681704	1.007255
1969	1186.073431	1.048084	742.526143	1.125431	116481.729420	25595.210382	1.043390
1970	1246.326555	1.105200	787.384744	1.158796	122417.145667	28390.826259	1.096701
1971	1306.341870	1.155646	829.734506	1.188481	129099.576274	30900.222791	1.143787
1972	1345.332072	1.256121	881.033111	1.252484	135557.588617	33136.347580	1.237570
1973	1333.648460	1.547125	949.399056	1.396185	140293.140793	34950.372353	1.469533
1974	1361.502722	2.070385	996.148083	1.644746	154033.835069	31823.505335	1.945106

YEAR	THIX	THII	THIC
1965	15387.000000	17143.000000	56920.000000
1966	18399.721206	21559.975586	62330.497524
1967	21032.570562	23478.180639	67505.223751
1968	22467.202883	26161.167148	73260.203924
1969	22945.232935	28125.495037	78692.211830
1970	23948.067877	29755.940002	83606.964047
1971	25140.177984	31591.570481	88396.050600
1972	27248.580426	33365.599982	93048.755789
1973	26882.500547	34666.976062	97168.036537
1974	27641.622092	38431.528990	103296.189322

E X O G E N O U S V A R I A B L E S

YEAR	WT$	PW$	PM$6	THIR	THIGP	XS$6	MS$6
1965	204586.437000	0.912999	0.911999	20.829986	8048.000000	136.457611	110.363968
1966	221354.437000	0.920999	0.924999	20.749984	8681.000000	252.462250	80.698150
1967	232835.562000	0.923999	0.922999	20.799987	9351.000000	355.180176	141.875549
1968	259325.000000	0.922999	0.926999	20.849990	11330.000000	390.477783	132.717499
1969	287506.000000	0.949999	0.954999	20.929992	12314.000000	371.298828	134.654099
1970	312508.500000	1.000000	1.000000	20.929992	13497.000000	375.120605	218.656357
1971	333437.750000	1.043999	1.036999	20.929992	14772.000000	390.639160	282.967285
1972	367282.187000	1.127999	1.133999	20.929992	15031.000000	441.688232	364.522461
1973	413530.500000	1.382999	1.409999	20.379989	16526.000000	355.551758	469.810791
1974	418280.636087	1.878671	1.822198	20.379989	16488.000000	345.652588	280.608398

$

A2.10 EXECUTION STATISTICS

```
SYS$SYSDEVICE:[SHI]JAPAN.ACT;2

MODULE; JAPAN
============
  Accounting information:
  Buffered I/O count:          294      Peak working set size:    234
  Direct I/O count:             85      Peak page file size:     4185
  Page faults:                 737      Mounted volumes:            0
  Elapsed CPU time:     0 00:00:28.13   Elapsed time:      0 00:07:47.87

SYS$SYSDEVICE:[SHI]KOREA.ACT;2

MODULE; KOREA
============
  Accounting information:
  Buffered I/O count:          294      Peak working set size:    282
  Direct I/O count:             74      Peak page file size:     4185
  Page faults:                 651      Mounted volumes:            0
  Elapsed CPU time:     0 00:00:16.13   Elapsed time:      0 00:07:39.70

SYS$SYSDEVICE:[SHI]PHILIPPI.ACT;2

MODULE; PHILIPPI
================
  Accounting information:
  Buffered I/O count:          290      Peak working set size:    287
  Direct I/O count:             78      Peak page file size:     3770
  Page faults:                 664      Mounted volumes:            0
  Elapsed CPU time:     0 00:00:20.40   Elapsed time:      0 00:07:36.35

SYS$SYSDEVICE:[SHI]TAIWAN.ACT;2

MODULE; TAIWAN
==============
  Accounting information:
  Buffered I/O count:          290      Peak working set size:    287
  Direct I/O count:            140      Peak page file size:     3770
  Page faults:                 663      Mounted volumes:            0
  Elapsed CPU time:     0 00:00:21.19   Elapsed time:      0 00:07:38.19

SYS$SYSDEVICE:[SHI]THAILAND.ACT;2

MODULE; THAILAND
================
  Accounting information:
  Buffered I/O count:          290      Peak working set size:    286
  Direct I/O count:             87      Peak page file size:     3770
  Page faults:                 655      Mounted volumes:            0
  Elapsed CPU time:     0 00:00:26.11   Elapsed time:      0 00:07:36.16

SYS$SYSDEVICE:[SHI]USA.ACT;2

MODULE; USA
===========
  Accounting information:
  Buffered I/O count:          295      Peak working set size:    281
  Direct I/O count:             76      Peak page file size:     4185
  Page faults:                 668      Mounted volumes:            0
  Elapsed CPU time:     0 00:00:18.28   Elapsed time:      0 00:07:41.78

SYS$SYSDEVICE:[SHI]WORLD.ACT;2

MODULE; WORLD
=============
  Accounting information:
  Buffered I/O count:         1288      Peak working set size:    469
  Direct I/O count:            130      Peak page file size:      414
  Page faults:                1361      Mounted volumes:            0
  Elapsed CPU time:     0 00:02:44.75   Elapsed time:      0 00:07:45.78
```

APPENDIX B

EBNF OF CONCURRENT MODEL

```
<MODELSPECIFICATION>::=[ <MODELBODYSTMTS> /CLRERRF/ ]*
     /STMTFL/  <MODELSPECIFICATION>  .
<MODELBODYSTMTS>::=    /E(80)/
   MODULE <MODULENAMESTMT>
  |SOURCE <SOURCEFILESSTMT>
  |TARGET <TARGETFILESSTMT>
  |MAXD : <INTEGER> /STMAX/ <ENDCHAR>
  | @ END @ /ENDINP/
    | <DCLDESCRIPTION>
    | <BLOCKBEGIN>
    | <BLOCKEND>
    | <OLDFILESTMT>
    | /ASSINIT/ <ASSERTIONS> /STRHS/
<DCLDESCRIPTION> ::= 1 /INTDCL/ /INTMVAR/ /MEMINIT/
                        /SVMEM/ <DATASPEC>
                     [, /E(108)/ <INTEGER> /CRDCL/ /INTMVAR/
                      /MEMINIT/ /SVMEM/ <DATASPEC> ]*
                      /STDCL/ <ENDCHAR>
<DATASPEC>    ::= <DCLMVAR> [( <OCCSPEC> )] [ <IS> ]
                  <ATTRSPEC> /SVDCL/
<ATTRSPEC>    ::= <FILE> /SVF/ /SVFLNM/ <FILEDESC> <STORAGEDESC>
/STDEV/
                  | <RECORD>                  /SVR/
                  | <FIELDSTMT> /STDFLD/ /SVD/
                  | [<GROUP>]                 /SVG/
<BLOCKBEGIN> ::= BLOCK /BLKINIT/ [ <NAME> /SVLBL/ ] /E(2)/ :
                       [ <BLOCKSPEC> ]* /SVBLOK/  <ENDCHAR>
<BLOCKSPEC>   ::= <SOLUTION> | <ITERATION> | <RELERROR>
<SOLUTION>     ::= [ SOLUTION ] METHOD [ <IS> ] /E(62)/ <METHODS>
                     /SVMETH/ [ , ]
<METHODS>      ::=  NEWTON | GAUSSSEIDEL | GS | JACOBI
<ITERATION>    ::=  [ <MAXIMUM> ] <ITER> [ <IS> ] /E(4)/ <NUMBER>
                     /SVITER/ [ , ]
<MAXIMUM>      ::=  MAX | MAXIMUM
<ITER>         ::=  ITER | ITERATION | ITERATIONS
<RELERROR>    ::= [ RELATIVE ] <ERROR> [ <IS> ] /E(5)/ <NUMBER>
                     /SVERR/ [ , ]
<ERROR>        ::=  ERR |  ERROR
<BLOCKEND> ::= <END> /BLKEND/ [ <NAME> /CHKLBL/ ]  <ENDCHAR>
<END>          ::= /ENDID/
<ASSERTIONS>::=/E(14)/<CONDITIONAL> |
                /SVASSR/ /INTMVAR/ <MVAR> /STMVAR/ /SVCMP1/
                [<IS>/SVNXOP/]<DDLORRHS>
<CONDITIONAL>::=IF /SVAAS1/ /SVOP1/ /SETBIT/ /E(18)/
                <BOOLEANEXPRESSION> /SVCMP1/ /E(38)/
                THEN /SVNXOP/ <SIMPLEASSERTION> /SVNXCMP/
                [ELSE /SVNXOP/ <ASSERTION> /SVNXCMP/] /RSETIF/ /STALL/
```

```
<ASSERTION>::= /E(14)/ <CONDITIONAL> | <SIMPLEASSERTION>
<DDLORRHS>::=/INTODDL/ <DATADESCSTMT> /FREETMP/
              | /E(33)/ <INTOAS> <ASSERTIONBRANCH> <ENDCHAR>
<ASSERTIONBRANCH>::= <DEFEXPRESSION>
                   | <BOOLEANEXPRESSION>/SVNXCMP/ /STALL/
<DEFEXPRESSION>::=  ( /INTSUB/ <VALUELIST> ) /FREESUB/
<VALUELIST>::= ( /CRSUB/ /DECPP/ <VALUELIST> [, <VALUELIST> ]* )
                /INCPP/
              | <OPELEMENT>  /STASS/
<OPELEMENT>::= [<SIGN> /SVOPP/] <NUMBER> /STNUM/
              | <STRINGFORM>
<INTOAS>::=/INTOASS/
<SIMPLEASSERTION>::=/SVASAE1/ /INTMVAR/ <MVAR> /STMVAR/ /CKUNIK/
                    /SVCMP1/ /E(23)/ = /SVNXOP/
                    <BOOLEANEXPRESSION> /SVNXCMP/ /STALL/ <ENDCHAR>
<SUBVARIABLE>::= /SETSUBV/ <VAR> /SVCKMV/ /SVCMP1/
        [(/SVNXOP/ /SETBIT/ /E(22)/
          <BOOLEANEXPRESSION> /SVNXCMP/ /SVCKSUB/ [,/SVNXOP/
          <BOOLEANEXPRESSION> /SVNXCMP/ /SVCKSUB/ ]*
        /E(24)/ ) ] /CLCKSUB/ /STALL/
<SUBVARIABLE1>::= /SETSUBV/ <VAR> /SVCMP1/
        [(/SVNXOP/ /SETBIT/ /E(22)/
          <BOOLEANEXPRESSION> /SVNXCMP/ [,/SVNXOP/
          <BOOLEANEXPRESSION>/SVNXCMP/]*
        /E(24)/ ) ] /STALL/
<BOOLEANEXPRESSION>::= /E(82)/ /SVBEXP/ <CONDEXP> |
                       <BOOLEANTERM> /SVCMP1/
                       [<OR> /SVNXOP/ <BOOLEANTERM> /SVNXCMP/]*
                       /STALL/
<CONDEXP>::=IF /SVCOND/ /E(3)/ <BOOLEANEXPRESSION> /SVCMP1/
             /E(79)/ THEN /SVNXOP/ <BOOLEANEXPRESSION> /SVNXCMP/
             [/E(12)/ ELSE /SVNXOP/ <BOOLEANEXPRESSION> /SVNXCMP/]
             /STALL/
<OR>::= /ORREC/
<BOOLEANTERM>::= /E(83)/ /SVBT1/ <BOOLEANFACTOR> /SVCMP1/
                 [_/SVNXOP/ <BOOLEANFACTOR> /SVNXCMP/]*
                 /STALL/
<BOOLEANFACTOR>::= /E(82)/ /SVBF1/ <CONCATENATION> /SVCMP1/
                   [<RELATION> /SVNXOP/ <CONCATENATION> /SVNXCMP/]*
                   /STALL/
<RELATION>::= /RELREC/
<CONCATENATION>::= /E(84)/ /SVCON/ <ARITHEXP> /SVCMP1/
                   [ <CONCAT> /SVNXOP/ <ARITHEXP> /SVNXCMP/]*
                   /STALL/
<CONCAT>::= /CATREC/
<ARITHEXP>::= /E(81)/ /SVAE/ [<SIGN> /SVOP1/]
              <TERM> /SVCMP1/ [<OPS> /SVNXOP/ <TERM> /SVNXCMP/]*
              /STALL/
<TERM>::= /E(87)/ /SVTERM/ <FACTOR> /SVCMP1/
          [<MOPS> /SVNXOP/ <FACTOR> /SVNXCMP/]* /STALL/
<FACTOR>::= /E(85)/ /SVFAC/ [ /SVOP1/] <PRIMARY> /SVCMP1/
            [<EXPON> /SVNXOP/ <PRIMARY> /SVNXCMP/]* /STALL/
<EXPON>::= /EXPREC/
<PRIMARY>::= /E(86)/ /SVPRIM/ <ISPRIM> /SVCMP1/ /STALL/
<ISPRIM>::= ( <BOOLEANEXPRESSION> /E(24)/ )
            | <NUMBER> /STNUM/
            | <STRINGFORM>
            | <FUNCTIONCALL>
            | <SUBVARIABLE1>
<STRINGFORM>::= ' /SETSTRN/ [ <STRING>  /SVSTRNG/]
                /E(26)/
                ' /ADLEX/ [B /STBIT/ /E(1)/ <BSUFX>] /STNUM/
<FUNCTIONCALL>::= <FUNCTIONNAME> /STFUN/
                  /SETFUNC/ [(/SVNXOP/ <BOOLEANEXPRESSION>
                  /SVNXCMP/ [,/SVNXOP/ <BOOLEANEXPRESSION>
                  /SVNXCMP/ ]* ) ] /STALL/
```

```
<FUNCTIONNAME>::= /FNCHECK/
<MVAR>::=  ( <SUBVARIABLE> /SVMVAR/
           [, <SUBVARIABLE> /SVMVAR/ ]* /E(24)/ )
           | <SUBVARIABLE> /SVMVAR/
<VAR>::= /SETVAR/ /INITQNM/ /E(68)/ <NAME> /ADLEX/ /MKQNM/
         [. /ADLEX/ /E(68)/ <NAME> /ADLEX/ /MKQNM/]* /STRCON/
<DCLMVAR> ::= ( <VAR> /SVCKMV/ /SVMVAR/
                [, <VAR> /SVCKMV/ /SVMVAR/ ]* )
              | <VAR> /SVCKMV/ /SVMVAR/
<BSUFX>::= /BITSTR/
<QNAME>::= /INITQNM/ /E(68)/ <NAME>  /MKQNM/
    [ .  /E(68)/ <NAME>  /MKQNM/ ] *
<STRING>::= <STRINGCONST>
<OPS>::= /OPREC/
<MOPS>::= /MOPREC/
<TEST>::= /TESTBIT/
<MODULENAMESTMT>::=       /E(63)/: /E(64)/ <NAME>
    /STMOD/ <ENDCHAR>
<SOURCEFILESSTMT>::=       [<FILEKEYWORD>] /E(75)/ /INITSFL/ :
    <SOURCEFILELIST>     /STSRC/ <ENDCHAR>
<FILEKEYWORD>::=FILES|FILE
<SOURCEFILELIST>::= /E(76)/ <NAME> /SVSRC/
    [, /E(76)/ <NAME> /SVSRC/]*
<TARGETFILESSTMT>::=       [<FILEKEYWORD>] /E(77)/ /INITTFL/ :
    <TARGETFILELIST>    /STTAR/ <ENDCHAR>
<TARGETFILELIST>::= /E(78)/ <NAME> /SVTAR/
    [, /E(78)/ <NAME> /SVTAR/ ]*
<DATADESCSTMT>::=  <DATADESCRIPTION> <ENDCHAR>
<DATADESCRIPTION>::=
  <FILESTMT>     /STFILE/
  |<RECORDSTMT> /STREC/
  |<GROUPSTMT>  /STGRP/
  |<FIELDSTMT> /STFLD/
  |<SUBSTMT>     /STSUBST/
<SUBSTMT>::=<SUBSCRIPT>/MEMINIT/ /SVMEM/ [( <OCCSPEC> )]
<SUBSCRIPT>::= SUB | SUBSCRIPT | SUBSCRIPTS
<FILE>::= FILE | REPORT | FILES | REPORTS
<RECORDSTMT>::= <RECORD> /MEMINIT/ [(] <ITEMLIST> [)]
<RECORD>  ::=  REC | RECORD | RECORDS
<ITEMLIST>::= /E(52)/<ITEM> [[,] <ITEM>]*
<ITEM>::=<NAME> /SVMEM / [ . <NAME> /SVMEM/ ]* [( <OCCSPEC> )]
<OCCSPEC>::=  <STAR> /SVSTAR/ | <MINOCC>/SVMNOC/ [<MAXOCC>]
<STAR>::= /STARREC/
<MINOCC>::=<INTEGER>
<MAXOCC>  ::=  [:/E(51)/]<INTEGER> /SVMXOC/ /CKMNMX/
               |          <INTEGER> /SVMXOC/ /CKMNMX/
<GROUPSTMT>::= <GROUP>/MEMINIT/ [(] <ITEMLIST> [)]
<GROUP>  ::=  GRP | GROUP | GROUPS
<FIELDSTMT>::=  <FIELD> /SVFLD/ <FIELDATTR>
                [<ONCND> [:] /E(112)/<OPT>/SVOP/]
<ONCND> ::= ONCNVERR | ONCERR
<OPT>    ::= STOP | <BINORNUM>
<BINORNUM> ::= '/E(110)/<BIN>'B/E(111)/ <CKBIN>
               | [<SIGN> /SVS/] <NUMBER>
<CKBIN> ::=/CKBIN/
<FIELD>  ::=  FLD | FIELD | FIELDS
<FIELDATTR>::= [(]  <TYPE> /SVFDTP2/[ <LENGSPEC>]
               [,]  [<LINESPEC>]  [,]  [<COLSPEC>]  [)]
<LENGSPEC> ::= ( /E(48)/ <MINLENGTH> [ <MAXLENGTH> ] /E(49)/ )
               |<MINLENGTH> [<MAXLENGTH>]
<MINLENGTH>::= <INTEGER> /SVMNFLN/
<LINESPEC>::= LINE /E(53)/ /E(54)/ /E(55)/ (<INTEGER> /SVLINE/)
<COLSPEC>::= COL /E(90)/ /E(91)/ /E(92)/ (<INTEGER> /SVCOL/)
<TYPE>::= /E(47)/ <PICDESC> | <STRINGSPEC>  | <NUMSPEC> | <GENERIC>
<GENERIC> ::= <GEN> /SVGEN/
<GEN>     ::= GENERIC | GEN
```

```
<PICDESC>::= <PICTYPE> /E(67)/ /SVPIC/ ' [ <STRING> /SVPICST/ ] '
             /STPIC/
<PICTYPE>::= PIC | PICTURE
<STRINGSPEC>::= <STRINGTYPE> /SVSTRTP/
<STRINGTYPE>::= CHAR | CHARACTER | BIT | NUM | NUMERIC
<NUMSPEC>::= <NUMTYPE> /SVNUMTP/ [ <FIXFLT> /SVMOD/ ]
<NUMTYPE>::= BIN | BINARY | DEC | DECIMAL
<FIXFLT>::= FIX | FIXED | FL | FLOAT | FLT
<MAXLENGTH>::= [:] <INTEGER> /SVMXFLN/
               | , /E(46)/ <SINTGR> /SVSCALE/
               | <INTEGER> /SVMXFLN/
<SINTGR>::= - /E(50)/ <INTEGER> /NEGATE/ | <INTEGER>
<NUMBER> ::= /SETNUM/ <INITNUM> /E(65)/ <RECNUM>
<RECNUM>::= /RECNUM/
<BIN> ::= /BIN/
<INITNUM>::= /INITNUM/
<SIGN>::= + | -
<RECG>::= <RECORD> | <GROUP>
<KEY>::=KEY|SEQUENCE
<CODE>::=EBCDIC|BCD|ASCII
<ANY>::= <NAME>|<INTEGER>
<NOTRKS>::= 7|9
<DENSITY>::= 200|556|800|1600|6250
<PARITY>::= ODD|EVEN
<TYPEDSK>::= 2314|2311|3330|2305 | 3330-1
<ORG>::=ORG|ORGANIZATION
<ORGTYPE>::= /E(7)/ISAM|SEQUENTIAL|SAM|INDEXEDSEQUENTIAL|POST|MAIL
<ENDCHAR>::= /E(74)/ <ENDCHAR> /STMTINC/
<ENDCHAR>::= /SVENDC/
<STRINGCONST>::=/CHARSTR/
<NAME>::=/NAMEREC/
<INTEGER>::=/INTREC/
<IS>::= IS | = | ARE
<FILESTMT>::=  <FILE> /SVFLNM/ /MEMINIT/ <SONDESC>
               <FILEDESC> <STORAGEDESC> /STDEV/
<SONDESC>::=( <ITEMLIST> )
             | <RECG> [NAME] [<IS>] [(] <ITEM> [)]
<OLDFILESTMT>::= <FILE> [NAME] [<IS>] /E(56)/ /MEMINIT/ /INTMVAR/
                 <DCLMVAR> /SVFLNM/
                 <RECG> [NAME] [<IS>] [(] <ITEM> [)]
                 <FILEDESC> /STFILE/
                 <STORAGEDESC> /STDEV/  <ENDCHAR>
<FILEDESC>::=   [STORAGE [NAME] [<IS>] /E(44)/ <NAME> /SVSTNM/]
                [<KEY> [NAME] [<IS>] /E(45)/ <NAME> /SVKEY/]
                [<ORG> [<IS>] <ORGTYPE> /SVORG3/]
<STORAGEDESC>  ::= [DEVICE [<IS>] <DEVICE>] /SVDEV/
        [RECORD /E(57)/][FORMAT [<IS>] <RECFMT>]/SVRECF/
        <BLKRECVOL>
        [<TAPEDESC>] [<DISKDESC>]
        [HARDWARE]  [SOFTWARE]
<DEVICE>  ::= /E(61)/ TAPE | DISK/SETDEVB/
      | CARD /SETDEVC/     | PRINTER /SETDEVP/
      | PUNCH /SETDEVU/    | TERMINAL /SETDEVT/
<RECFMT>  ::= /E(69)/ FIXED|VARIABLE|VARSPANNED|UNDEFINED
<BLKRECVOL>  ::=
        [ [MAX] /E(70)/ /E(71)/ BLOCKSIZE [<IS>] <INTEGER> /SVBLK/ ]
        [ [MAX/E(59)/] RECORDSIZE [<IS>] /E(72)/<INTEGER>/SVRCSZ/]
     [VOLUME [NAME] [<IS>] /E(60)/ <NAME>/SVVOL/ [,/E(60)/<NAME>]* ]
<TAPEDESC>  ::=  [<TRACKS> [<IS>] /E(66)/<NOTRKS>/SVTRK2/ ]
                 [PARITY [<IS>] /E(66)/ <PARITY>/SVPAR2/]
                 [DENSITY [<IS>] /E(66)/ <DENSITY> /SVDEN2/]
                 [ [TAPE] LABEL [<IS>] <LABELTYPE>/SVLAB2/]
                 [START [FILE] [<IS>] /E(66)/ <INTEGER> /SVSTFL2/]
                 [[CHAR] CODE [<IS>] <CODE> /SVCC/ ]
<TRACKS>  ::=  NOTRKS | TRACKS
<LABELTYPE>  ::= /E(58)/ IBMSTD|ANSISTD|NONE|BYPASS
```

```
<DISKDESC>   ::=  [UNIT [<IS>] /E(9)/ <TYPEDSK> /SVUNIT2/]
                   [<CYLINDERS>/SVUCYL/ [<IS>] /E(66)/ <INTEGER>
                    /SVQTY2/]
<CYLINDERS>  ::=  NOCYLS | CYLINDERS
<HARDWARE>::= [[COMPUTER] MODEL [<IS>] <ANY>
<SOFTWARE>::= [[OPERATING] SYSTEM [<IS>] <ANY>]
```

APPENDIX C

WARNING AND ERROR MESSAGES GENERATED BY THE CONFIGURATOR

C1. ERROR/WARNING MESSAGES FOR SYNTAX ANALYSIS:

1. *WARNG* (LEX1) THE IDENTIFIER "text..." IS TRUNCATED TO 10 CHARACTERS.
2. *ERROR* (LEX2) THE LAST STATEMENT DOES NOT TERMINATE WITH "*/".
3. *ERROR* (LEX3) THE LAST STATEMENT DOES NOT TERMINATE WITH ";"
4. *ERROR* (LEX4) CANNOT FIND THE NAMED CONFIGURATION INPUT FILE.
5. *ERROR* (SAP1) MISSING ';' AT END OF A STATEMENT. STMT: stmt#
6. *ERROR* (SAP2) ILLEGALLY STRUCTURED NAMES FOUND. STMT: stmt#
7. *ERROR* (SAP3) ILLEGALLY STRUCTURED STATEMENT. STMT: stmt#
8. *ERROR* (SAP4) MISSING ':' AFTER 'F' OR 'M'. STMT: stmt#
9. *ERROR* (SAP5) ILLEGAL NAME. STMT: stmt#
10. *ERROR* (SAP6) ILLEGAL FILE ORGANIZATION. STMT: stmt#
11. *ERROR* (SAP7) RECORD SIZE MUST BE AN INTEGER!. STMT: stmt#
12. *ERROR* (SAP8) VERSION NUMBER MUST BE AN INTEGER!. STMT: stmt#
13. *ERROR* (SAP9) ILLEGAL NAME FOUND IN A SYNONYM STATEMENT. STMT: stmt#
14. *ERROR* (SAP10) MISSING ',' IN BETWEEN TWO NAMES IN A SYNONYM STATEMENT. STMT: stmt#
15. *ERROR* (SAP11) MISSING TAGET NAME OF AN ARROW IN A PATH STATEMENT
16. *ERROR* (SAP12) ILLEGAL KEYWORD. EXPECT: "TYP","ORG", OR "TYPE"

C2. ERROR/WARNING MESSAGES FOR CONFIGURATION GRAPH CONSTRUCTION:

17. *ERROR*(CNF1) THE SYNONYM NAMES HAVE NO PHYSICAL NAME DEFINED:

----	...NAME...	.ORG.	.RECSZ.	..FULLPHYSICALNAME.....
	name1	org1	rec.size1	pname1
	name2	org2	rec.size2	pname2
	name3	org3	rec.size3	pname3
	:	:	:	:

18. *ERROR*(CNF2) CONFLICT ATTRIBUTE(S) FOUND FOR SYNONYM FILES:

----	...NAME...	.ORG.	.RECSZ.	..FULLPHYSICALNAME.....
	name1	org1	rec.size1	pname1
	name2	org2	rec.size2	pname2
	name3	org3	rec.size3	pname3
	:	:	:	:

19. *ERROR*(CNF3) AN M->M PATTERN FOUND IN LINE: stmt_no
20. *ERROR*(CNF4) A MODULE NAME CANNOT BE SUFIXED.
    ILLEGAL NAME: name
21. *ERROR*(CNF5) ILLEGAL MODULE NAME. THE NAME HAS BEEN USED FOR A FILE. name
22. *ERROR*(CNF6) A REDUNDANT AND CONFILICTING STATEMENT FOUND FOR

```
          MODULE:  name, THE CONFILICTING PHYSICAL NAMES:
          name1,name2, STMT: stmt#
23.*ERROR*(CNF7) A REDUNDANT AND CONFILICTING STATEMENT POUND FOR
          MODULE:  name. THE CONFILICTING LOCATIONS:
          location1,location2, STMT: stmt#
24.*ERROR*(CNF8) A REDUNDANT AND CONFILICTING STATEMENT POUND FOR
          MODULE: name, THE CONFILICTING DEVICES:
          device1,device2,STMT: stmt#
25.*ERROR*(CNF9) A REDUNDANT AND CONFILICTING STATEMENT POUND FOR
          MODULE: name. THE CONFILICTING DIRECTORIES:
          directory1,directory2,STMT: stmt#
26.*ERROR*(CNF10) A REDUNDANT AND CONFILICTING STATEMENT FOUND FOR
          MODULE: name. THE CONFILICTING SUFIXES:
          sufix1,sufix2,STMT: stmt#
27.*ERROR*(CNF11) A REDUNDANT AND CONFILICTING STATEMENT FOUND FOR
          MODULE: name. THE CONFILICTING VERSIONS:
          version1, version2, STMT: stmt#
28.*ERROR*(CNF12) A REDUNDANT AND CONFILICTING STATEMENT FOUND FOR
          MODULE: name. THE CONFILICTING ORGANIZATIONS:
          org1,org2,STMT: stmt#
29.*ERROR*(CNF13) A REDUNDANT AND CONFILICTING STATEMENT FOUND FOR
          MODULE: name. THE CONFILICTING STATUSES:
          status1, status2, STMT:stmt#
30.*ERROR*(CNF14) ILLEGAL FILE NAME. THE NAME HAS BEEN USED FOR A
          MODULE: name, STMT: stmt#
31.*ERROR*(CNF15) A REDUNDANT AND CONFILICTING STATEMENT FOUND FOR
          FILE: name. THE CONFILICTING PHYSICAL NAMES:
          pname1,pname2,STMT: stmt#
32.*ERROR*(CNF16) A REDUNDANT AND CONFILICTING STATEMENT FOUND FOR
          FILE: name. THE CONFILICTING LOCATIONS:
          location1,location2,STMT: stmt#
33.*ERROR*(CNF17) A REDUNDANT AND CONFILICTING STATEMENT FOUND FOR
          FILE: name. THE CONFILICTING DEVICES:
          device1,device2, STMT: stmt#
34.*ERROR*(CNF18) A REDUNDANT AND CONFILICTING STATEMENT FOUND FOR
          FILE: name. THE CONFILICTING DIRECTORIES:
          directory1, directory2, STMT: stmt#
35.*ERROR*(CNF19) A REDUNDANT AND CONFILICTING STATEMENT FOUND FOR
          FILE: name. THE CONFILICTING SUFIXES:
          sufix1, sufix2, STMT: stmt#
36.*ERROR*(CNF20) A REDUNDANT AND CONFILICTING STATEMENT POUND
          FOR FILE: name. THE CONFILICTING VERSIONS:
          version1, version2, STMT: stmt#
37.*ERROR*(CNF21) A REDUNDANT AND CONFILICTING STATEMENT FOUND FOR
          FILE: name. THE CONFILICTING ORGANIZATIONS:
          org1, org2, STMT: stmt#
38.*ERROR*(CNF22) A REDUNDANT AND CONFILICTING STATEMENT FOUND FOR
          FILE: name. THE CONFILICTING RECORD SIZES:
          rec.size1, rec.size2, STMT: stmt#
39.*WARNG*(CNF23) PHYSICAL NAME TYPE LENGTH EXCEEDS 3 (type).
          TRUNCATED.
40.*ERROR*(CNF24) UNDEFINED NAME POUND IN A SYNONYM STATEMENT:
           name, STMT: stmt
41.*ERROR*(CNF26) ILLEGAL SYNONYM STATEMENT (NO DEFINED SYMBOL
           HAS BEEN FOUND). STMT: stmt#
42.*ERROR*(CNF27) DIFFERENT NODE TYPES SPECIFIED FOR SYNONYM NAMES:
      ------ ...NAME... .TYPE.
                name1     type1
                name2     type2
                  :         :
                  .         .
43.*ERROR*(CNF28) WRONG STATEMENT ORDER. NO PATH STATEMENT
          CAN BE PLACED AFTER SYNONYM(S).
```

C3. ERROR/WARNING MESSAGES DURING COMPLETENESS ANALYSIS

```
44.*ERROR* (CMP1) AN ISOLATED FILE NODE FOUND: name
45.*WARNG* (CMP2) INCOMPLETE DEFINITION FOUND FOR A
           "MAIL" OR "POST" FILE: name
46.*ERROR* (CMP3) MORE THAN ONE PRODUCER FOUND FOR A POST
           FILE: name
47.*ERROR* (CMP4) MORE THAN ONE PRODUCER FOUND FOR A SAM
           FILE: name
48.*ERROR* (CMP5) AN ISOLATED MODULE NODE FOUND: name
49.*ERROR* (CMP6) MORE THAN ONE PRODUCER OR CONSUMBER
           FOUND FOR A "MAN" MODULE: name
50.*ERROR* (CMP7) A "MAN" MODULE MUST COMMUNICATE WITH
           "MAIL" FILES. (name)
51.*ERROR* (CMP9) THE FILE(S) CONSUMED OR PRODUCED BY
           MODULE: name ARE NOT CO-LOCATED.


C4. ERROR/WARNING MESSAGES DURING SCHEDULING AND DOCUMENTATION
    GENERATION

52.*ERROR* (RPT2) MORE THAN ONE PRODUCER FOUND FOR A SEQ
           FILE "name".
53.*WARNG* (RPT3) MORE THAN ONE CONSUMER FOUND FOR A SEQ
           FILE "name". THEY ARE SCHEDULED SEQUENTIALLY.
54.*ERROR* (RPT5) MORE THAN ONE CONSUMER FOUND FOR A MAIL
           FILE "name".
55.*ERROR* (RPT6) MORE THAN ONE PRODUCER FOUND FOR A POST
           FILE "name".
56.*WARNG* (RPT7) A MULTI-NODE MAXIMALLY STRONGLY CONNECTED
           COMPONENT FOUND IN CONFIGURATION CONSISTS OF:
           - element name1
           - element name2
                 :
                 .
           THE FILE NODES MUST HAVE A VIRTUAL DIMENSION.
57.*ERROR* (SEQ1) A MAXIMALLY STRONGLY CONNECTED
           COMPONENT CONTAINS SEQUENTIAL
           EDGES:
           - Component 1: ele1<SEQ>, ele2, ...
           - Component 2: ele1, ele2<SEQ>, ...
           - Component 3: ele1, ele2, ...
                 :
                 .
                 :
58.*ERROR* (SCD1) CYCLES FOUND IN A SCHEDULE GRAPH
           CONSISTING OF:
           - PAR NODE: ele1, ele2<SEQ>,...
           - PAR NODE: ele1, ele2,...
           - PAR NODE: ele1<SEQ>, ele2,...
                 :
                 .
```

Note.

Every message is associated with a unique code which is used to identify the producer of the message. The following list shows the correspondence between the codes and programs.

Code	Program name	Description
CNF	CONF	Main Controler
LEX	LEX	Lexical Analyzer
SAP	SAP	Syntax Analyzer
CMP	CMPANA	Completeness Checking
SEQ	SEQCK	Consistency Checking
SCD	SCHEDULE	Scheduling
RPT	GRPT	Documentation

BIBLIOGRAPHY

1. Ackerman, W.B., "Data Flow Languages", Computer, Vol.15, No.2, pp. 15-25, February 1982.

2. Aho, A.V., Hopcroft, J.E., and Ullman, J.D., "The Design and Analysis of Computer Algorithms", Reading, Mass., Addison-Wesley, 1974.

3. Allen, James F., "Mantaining Knowledge about Temporal Intervals", Communications of the ACM, pp. 832-843, November 1983.

4. Andrews, G.R. amd Schineider,F.B., "Concepts and Notations for Concurrent Programming", Computing Survey, Vol.15, No.1, pp. 3-43, March 1983

5. Arvind, Culler, D.E., Iannuci, R.A., Kathail, V., Pingali, K. and Thomas, R.E., "The Tagged Token Dataflow Architecture," Laboratory for Computer Science, MIT, August 8, 1983.

6. Arvind, Gostelow, R.P. and Plouffe, W., "An Asynchronous Programming Language and Computing Machine," Dept. of Information and Computer Science Report 114a, University of California-Irvine, Dec. 1978.

7. Backus, J., "Can Programming Be Liberated From the von Neumann Style ? A Functional Style And Its Algebra Of Programs," Comm of the ACM, Vol.21, pp. 613-641, August 1978.

8. Berztiss,A.T., "Data Structures, Theory and Practice". Academic Press, 1971.

9. Brinch, H.P., "Distributed Processes-A Concurrent Programming Concept", CACM, Vol.21, No.11, pp. 934-941, Nov. 1978.

10. Brooks,S.D., "On the Relationship of CCS and CSP", Report CMU-CS-83-111, Department of Computer Science, Carnegie-Mellon University, March 1983.

11. Chen, B-S and Yeh, R.Y., "Formal Specification and Verification of Distributed Systems," IEEE Trans. on Software Engineering, Vol.SE-9, No.6, pp. 710-722, Nov. 1983.

12. Cheng, T., Lock, E. and Prywes, N., "Use of Program Generation by Accountants in the Evolution of Accounting Systems," Proc. of the Workshop On Reusability In Programming, ITT Programming, September 1983.

13. Coffman,E.G.,Jr., Elphick, M.J. and Shoshani, A., "System Deadlocks", Computing Surveys, Vol.3, No.2, pp.67-68, June 1971.

14. Dennis, J.B., "Data Flow Supercomputers," Computer, Vol.13, No.11, pp.48-58, Nov. 1980.

15. Dijkstra, E.W., Feijeu, W.H.J. and van Gasteren, A.Y.J., "Deriviation of a Termination Detection Algorithm for Distributed Computations," Information Processing Letters, Vol. 16, No.5, pp. 217-219, 1983.

16. Francez, N. and Rodeh, M., "Achieving Distributed Termination Without Freezing," IEEE Trans. on Software Engineering, Vol.SE-8, No.3, pp. 287-292, May 1982.

17. Freedman, A.L., and Lees, R.A., "Real Time Computer Systems", Computer-systems Engineering Series, Crane-Russak Co., 1977.

18. Gajski, D.D., Padua, A., Kuck, D.J., "A Second Opinion on Data Flow Machines and Languages," Computer, Vol.15, No.2, pp. 58-68, Feb. 1982.

19. Gana, J.L., "An Automatic Program Generator For Model Building In Social and Engineering Science", Dissertation in Computer Science, University of Pennsylvania, October 1978.

20. Gard, J. and Watson, I.,"Data Driven System For High Speed Parallel Computing," Computer Design, Vol.19, No.6 and 7, June and July 1982.

21. Gokhale, M.B., "Generating Data Flow Programs From Nonprocedural Specifications", Ph.D Dissertation, Department of Computer Science, University of Pennsylvania, March 1983.

22. Greenberg, R.G., "Simultaneous Equations In The MODEL System With An Application To Econometric Modelling", Master Thesis, University of Pennsylvania, October 1981.

23. Harel, D., Nehab, S., "Concurrent AND/OR Programs", Report CS82-09, Department of Applied Mathematics, The Weizmann Institute, 1982.

24. Hansen, P. Brinch, "Distributed Processes A Concurrent Programming Concept", CACM, Vol. 21, 11, 1978, pp. 934-941.

25. Henson, P.B. "EDISON - a Multiprocessor Language", Software-Practice and Experience, Vol.11, pp.325-361, 1981

26. Henson, P.B. "The Design of Edison", same as above, pp. 363-396, 1981

27. Henson, P.B. "Edison Programs", same as above, pp.397-414, 1981

28. Hilfinger, P.N., "Abstraction Mechanism in ADA", ACM Distinguished Dissertation Series, MIT, 1982.

29. Hoare, C.A.R., "Communicating Sequential Processes", CACM, Vol.21, No.8, pp. 666-677, August 1978.

30. Hoare, C.A.R., "A Calculus Of Total Correctness For Communicating Processes", Science of Computer Programming, Vol.1, pp. 49-72, 1981.

31. Hoffman, C.M. and O'Donnell, M.J., "Programming With Equations," ACM Trans. On Programming Languages and Systems, Vol.4, No.1, pp. 83-112, January 1982.

32. Holmström, Sören, "PFL: A Functional Language for Parallel Programming", Report, Programming Methodology Group, Chalmers University of Technology and University of Göteborg, 1983.

33. Holt, R.C., Graham, G.S., Lazowska, E.D., Scott, M.A., Structured Concurrent Programming With Operating Systems Applications, Addison Wesley, 1978.

34. Holt, R.C., "Comments On Prevention of System Deadlocks", Comm.ACM, 14, No.1, pp.36-38, Jan. 1971.

35. Holt, R.C., "Some Deadlock Properties of Computer Systems", ACM Computing Survey, Vol.4, No.3, pp.179-196, Sept. 1972.

36. Jensen,P.A., and Barnes,J.W., "Network Flow Programming", John Woley & Sons, 1980.

37. Keller,R.M., "Formal Verification of Parallel Programs".Communications of the ACM 19,7, pp.371-384, July 1976.

38. Kernighan, B.W. and Pike,R., "The UNIX Programming Environment", Englewood Cliffs, NJ:Prentice-Hall, 1984.

39. Kernighan, B.W., "The UNIX System and Software Reusability", IEEE Trans. On Software Eng., Vol. SE-10, Number 5, pp. 513-518, 1984.

40. Klein, L.R., "The LINK Model of World Trade with Application to 1972-1973", in Kenen, P., ed., Quantitive Studies of International Economic Relations, Amsterdam, North Holland, 1975.

41. Kwong,Y.S. "On the Absence of Livelock in Parallel Programs". Lecture Notes in Computer Science, vol.70,: Semantic of Concurrent Computation, Springer-Verlag, pp. 172-190, New York, 1979.

42. Lamport, L., "Specifying Concurrent Program Modules", ACM Trans. On Programming Languages and Systems, Vol.5, No.2, pp. 190-222, April, 1983.

43. Laner, P.E., Torrigiani, P.R. and Shields, M.W., "COSY - A System Specification Language Based On Paths and Processes," Acta Informatica Vol.12, pp. 109-158, 1979.

44. Ledgard, H. "ADA An Introduction, ADA Reference Manual", Springer-Verlag, July 1980

45. Levin,G.M., and Gries,D. "A Proof System for Communicating Sequential Processes", Acta Inf. 15, pp. 281-302, 1981.

46. Liskov, B. and Scheifler, R., "GUARDIAN and ACTION: Linguistic Support for Robust Distributed Programs", ACM Transactions on Programming Languages and Systems, Vol.3, No.3, pp. 381-404, July 1983

47. Lu, K.S., "Program Optimization Based on a Non-Procedural Specification," Ph.D. Dissertation, Department of Computer Science, University of Pennsylvania, 1981.

48. MacQueen,D.B., "Models for Distributed Computing", Rapport de Recherche No. 351, Institut de Rechereche d'Informatique et dAutomatique, LeChesnay France, April 1979.

49. Manna, Z., Mathematical Theory of Computation, McGraw Hill, 1974.

50. McGraw, J.R., "The VAL Language: Description and Analysis," ACM Trans. On Programming Languages and Systems, Vol.4, No.1, pp. 44-82, Jan. 1982.

51. Milner, R., A Calculus Of Communicating Systems, Lecture Notes on Computer Science, Vol. 92, Springer Verlag, 1980.

52. Milner,R., "Flowgraphs and Flow Algebras", J.ACM 26, 4, 1979

53. Milner,R. and Milne, G. "Concurrent Porcesses and Their Syntax", J.ACM, 26, 2, 1979

54. O'Donnell, M.J., "Computing in Systems Described by Equations", LNCS, Vol.58, 1978.

55. Ossefort, Marty, "Correstness Proofs of Communicating Processes: Three Illustrative Examples from the Literature". ACM Tran. on Programming Languages and Systems". vol 5,4, pp. 620-640, October, 1983.

56. Parnas, D., "Designing Software For Ease of Extension and Contraction," IEEE Trans. on Software Engineering, SE-5, No.3, pp. 128-138, March 1979.

57. Pnueli, A., Prywes, N., "Distributed Processing In the MODEL System With An Application To Econometric Modelling", Design Report, Department of Computer Science, University of Pennsylvania, June 1981.

58. Pnueli, A., Prywes, N., and Zarhi,R. "Scheduling Equational Specifications And Non Procedural Programs", Automatic Program Construction Techniquess, Ch.13, MacMillan, 1983.

59. Pnueli, A., "The Temporal Semantics of Concurrent Computation," in Lecture Notes on Computer Science, Vol.70, Springer Verlag, pp. 1-20, 1979.

60. Prywes, N. "Distributed Large Scale Computation With an Application to World-Wide Econometric Studies", Large Scale Systems, Vol 7, North-Holland, 1984.

61. Prywes, N. and Pnueli, A., "Automatic Programming In Distributed Cooperative Computation" to appear in IEEE Trans. On Systems, Man and Cybernetics, Jan. 1984.

62. Prywes, N. and Pnueli, A., "Compilation of Nonprocedural Specifications Into Computer Programs", IEEE Transactions on Software Engineering, Vol.SE-9, No.3, pp. 267-279, May 1983.

63. Prywes, N., Pnueli, A., and Shastry, S., "Use of a Non-Procedural Specification Language and Associated Program Generator in Software Development", ACM Trans. Programming Languages and Systems, Vol.1, No.2, pp. 196-217, October 1979.

64. Prywes, N., Szymanski, B. and Shi, Y., "Nonprocedural-Dataflow Specification of Concurrent Programs," Proc. of COMPSAC83, pp. 287-294, November 1983.

65. Ramamrithan, K. and Keller, R.M., "Specification of Synchronizing Processes," IEEE Trans. On Software Engineering, Vol.SE-9, No.6, pp. 722-733, Nov. 1983.

66. Shapiro, E., "Object Oriented Programming in Concurrent Prolog", New Generation Computing. OHMSHA. LTD and Springer-Verlag, pp. 25-48, 1983.

67. Shaw, A.C., "The Logical Design of Operating Systems", Prentice-Hall, Ed. George Forsythe, 1974.

68. Smith, B.J., "A Pipelined, Shared Resource MIMD Computer," Proc. of the 1978 International Conference on Parallel Processing, p. 6-8, 1978.

69. Szymanski, B. and Prywes, N., "Efficient Handling Of Data Structures In Definitional Languages," submitted for publication, 1984.

70. Szymanski, B., Prywes, N., Lock, E. and Pnueli, A., "On the Scope of Static Checking In Definitional Languages," Proceedings of ACM'84 Annual Conference, The Fifth Generation Challenge, 1984.

71. Taylor, R.N., "Complexity of Analyzing Concurrent Porgrams", Dept. of Computer Science, University of Victoria, Technical Report #DCS-9-IR, May 1981.

72. Taylor, R.N. "A General-Purpose Algorithm for Analyzing Concurrent Programs", CACM, pp.362-376, May 1983.

73. Teichreow, D. and Hershey III, E.A., "PSL/PSA: A Computerized Technique For Structured Documentation and Analysis of Information Processing Systems," IEEE Trans. on Software Engineering, Vol.SE-3, No.1, pp. 41-48, Jan. 1977.

74. Topor, R.W., "Termination Selection for Distributed Computing," Information Processing Letters, Vol.18, No.1, pp. 33-36, Jan. 1984.

75. Trelearan, P.C., Brownbridge, D.R., Hopkins, R.P., "Data-Driven and Demand-Driven Computer Architecture," Computing Surveys, Vol.14, No.1, pp. 93-138, March 1982.

76. Tseng, J., "Very High Level Dataflow Programming of Real Time Systems," Ph.D. Dissertation Proposal, Department of Computer Science, University of Pennsylvania, 1983.

77. VAX/VMS SYSTEM SERVICES REFERENCE MANUAL, Order No. AA-D018B-TE, Digital Corperation, Maynard, Massachusetts, March, 1980.

78. VAX-11 PL/I ENCYCLOPEDIC REFERENCE, Order No. AA-H9525A-TE, Digital Corperation, Maynard, Massachusetts, March, 1980.

79. Wirth, N., "Toward a Discipline of Real-Time Programming", CACM, Vol.20, No.8, pp.577-583, August 1977

80. Zave, P., "An Operational Approach to Requirements Specification for Embedded Systems," IEEE Trans. On Software Engineering, Vol.SE-8, No.3, pp. 250-269, May, 1982.

81. Zave, P., "The Operational Versus The Conventional Approach To Software Development," CACM, Vol.27, No.2, pp. 104-118, Feb. 1984.

INDEX

Address, 24, 73 to 74, 76, 90, 134, 140
Array graph, 46 to 50, 60, 129, 132, 136, 144, 156 to 157, 160, 162
Assumptions, 9 to 10, 147
ATOMIC module, 72, 99
Attachment, 166 to 168

Basic connections, 102 to 103, 106
Block structure, 157 to 158, 162, 168

Circular, 12, 48 to 49, 61, 104, 107, 112, 124
CLOSE process, 138
Code generation, 28, 32, 84, 85, 87, 118, 129, 133, 136, 144, 166 to 167
Compatibility, 4, 22, 30 to 31, 38 to 39, 76, 79 to 80
Completeness, 4, 9, 20, 30, 47 to 48, 66, 85, 98, 124
Component graph, 31, 48, 124
Concurrency, 14 to 15, 17, 22, 31, 77, 79, 113, 121
Concurrent computation, 2, 9, 17, 19, 53, 63
Concurrent update, 128, 141
Configuration, 7, 9 to 10, 21, 21, 23 to 25, 27 to 32, 33, 46, 56, 58, 64 to 65, 66 to 67, 68 to 72, 77, 79 to 80, 82 to 84, 85, 87 to 89, 91 to 92, 98 to 104, 107, 111 to 113, 115, 118 to 124, 127, 152, 155
Configuration graph, 25, 27, 31 to 32, 69, 80, 84, 85, 87 to 89, 91 to 92, 98, 102, 115, 124
CONFIGURATOR, 4 to 5, 9 to 10, 14 to 15, 19 to 20, 22, 25 to 26, 27 to 32, 56, 64, 66, 66 to 67, 69, 72 to 73, 77, 83 to 84, 85 to 88, 115, 118, 120 to 122, 124 to 125, 127, 137, 152, 155, 167
Consistency, 4, 9, 30 to 31, 47 to 50, 66, 72, 103 to 104, 107, 111 to 112, 114, 124
Consumer, 21 to 24, 28, 31, 40, 71, 76, 79 to 80, 98 to 99, 102, 128
Control field, 169
Cooperative, 2, 11 to 13, 49, 53, 58
CSL, 20, 32, 66 to 67, 68 to 69, 72 to 73, 76, 78 to 84, 85, 88 to 89, 91 to 92, 94 to 96, 99, 118, 123 to 124, 155
Cutting, 159 to 161

Data structure, 34, 41, 74, 80, 108, 116, 135 to 136, 164, 166
Data structures, 4, 35, 39, 80 to 81, 97, 106, 114, 164
Dataflow analysis, 99
Deadlock, 11, 27, 50
Dependency, 4, 6, 37 to 38, 43, 50, 56, 60, 104, 128, 131, 131 to 132, 157, 160
DEPENDS_ON, 38, 50, 131 to 133, 156, 158 to 162, 166, 168
Diameter, 28, 32, 62, 68 to 69, 85, 106 to 107, 115 to 116, 118 to 119, 122, 148 to 149, 152 to 155, 167 to 168
Dining philosopher, 21
Direct configuration graph, 92
Distributed termination, 28, 32, 61, 146 to 147, 152
Documentation, 9, 27, 31, 67, 85, 87, 124
Dse, 146, 155 to 157, 162 to 163, 167 to 170

EBNF, 88 to 90, 92 to 93, 95, 136
Endfile, 140 to 141, 142
Error message, 60, 83, 89, 94 to 95, 107, 114
Error messages, 27 to 28, 89, 99, 107, 114
Extended component graph, 124
External dependency, 6, 43, 50, 56, 60, 104, 131, 131 to 132, 157

File organization, 134, 136

Finding diameter, 116, 155
Flowchart, 136, 144,
157 to 158, 160, 162 to 166

GAUSS-SEIDEL, 60, 129, 157, 167
GRP, 23, 71 to 72, 79, 100,
119 to 123, 166

Inconsistency, 31, 49, 104,
107, 112, 114, 124
Individual JCL, 83, 87,
118 to 119, 121 to 122, 167
Initialization, 113, 152, 154,
158, 161 to 162, 164, 168
Inter-module, 16, 30, 128 to 129
Invariant, 7, 149 to 151

JACOBI-LIKE, 129

LINK, 13, 56, 63, 97,
105 to 106, 121
Lock, 11, 23, 27, 33, 35, 38,
50, 143, 145, 157 to 166,
168 to 170
Logical name, 73 to 74, 76, 90,
115, 138, 140, 167

Mail and post files, 15, 51,
77, 80, 99, 128
Mailbox creation, 83, 87, 121,
123, 137
Mailbox deletion, 83, 87, 124
Main JCL, 83, 87, 119 to 120,
137
Manual initiation, 81
MODEL, 3 to 5, 9 to 11,
13 to 20, 22 to 24, 27, 29,
33, 36 to 37, 40 to 41, 46,
46, 48 to 52, 53 to 54,
56 to 58, 60 to 61, 63 to 64,
72, 76, 80 to 81, 89, 104,
119, 126, 127 to 130,
132 to 133, 134 to 136, 138,
143 to 144, 146, 157,
159 to 160, 167
Model compiler, 4 to 5,
9 to 10, 15, 20, 23 to 24,
27, 29, 46, 46, 48 to 50, 52,
56, 60 to 61, 63, 72, 104,
126, 127 to 130, 136, 143,
146, 157, 159
Modifications, 4, 126 to 127,
129, 157

Non-circular, 107

ON-UNIT, 145
OPEN process, 136 to 137
Operating system, 3, 51, 73,
77, 118

Parameter, 55, 68, 83 to 84
Parser, 88, 95
Path, 9, 66, 68 to 71,
75 to 76, 80, 92 to 95, 109,
112, 115 to 116, 148, 150, 153
Physical, 10, 24 to 25,
70 to 74, 76 to 77, 82, 87,
90, 119, 137 to 138, 140,
164, 166
Physical name, 25, 72 to 74,
76 to 77, 90
Prepare, 144, 163
Producer, 21 to 24, 28, 31, 40,
71, 76, 79 to 81, 98 to 99,
102, 128, 134, 140
Program generation, 6, 28,
47 to 48, 64, 120 to 121, 123
PROLOG, 14, 17, 64

Query sub-system, 64

READ process, 138 to 139
Real time systems, 2, 4, 11
Recalculation, 168 to 169
Reduced econometric model,
53 to 54

Scheduling, 4, 27, 48, 67, 72,
85, 99 to 100, 111, 113, 129,
133, 136, 143 to 144, 157,
163, 168
Semantic routine, 89, 136
Sequencing, 103, 107
Sequentiality, 75, 79 to 80
SFS, 108 to 110
Simultaneous equations, 27,
51 to 52, 56, 60, 69, 115,
128 to 129, 145, 146, 148,
156, 161, 167 to 168
SOURCE, 11 to 12, 16, 21 to 25,
33, 33, 35, 38, 40, 42 to 43,
47, 49 to 50, 54 to 55, 58,
63, 76, 78, 123, 127 to 128,
131, 136 to 138, 141,
159 to 160, 162 to 163
Specification, 3 to 4, 6 to 7,
9, 14 to 15, 17 to 18, 21,
23, 25, 27, 29, 32, 33, 35,
37 to 38, 46 to 48, 51,
53 to 54, 56, 63 to 64,

66 to 67, 68, 72, 76, 79,
81 to 84, 85, 89, 92,
95 to 96, 99, 101, 118,
127 to 128, 143 to 144, 155,
157, 159 to 160
Specification languages, 3
Sublinear, 36, 49, 164
SUBMIT, 12, 118 to 119,
121 to 122
Subscript, 24, 35, 41, 47, 49,
56, 131
Symbol table, 95, 98
Synchronization, 18, 29, 76,
119, 121 to 122
SYNCHRONIZE, 29, 119,
121 to 122, 148
SYNONYM, 70, 82, 91, 97
Syntax analyzer, 88
Synthesis, 7, 9, 11, 13, 59

TARGET, 21 to 22, 33, 38,
54 to 55, 58, 77 to 78, 106,
127 to 128, 131, 136,
138 to 139, 158 to 160,
162 to 163
Temporal relations, 30 to 31,
100 to 101, 103, 111
Termination control algorithm,
115, 147, 167 to 168
Token, 61, 148 to 149,
151 to 154, 163, 166,
168 to 169
Top-down, 8 to 9, 11 to 12
Topological sorter, 160

Unravelling, 158, 161 to 162

VAX/VMS, 3, 28 to 29, 32, 73,
118
Vector, 34 to 36, 38, 40 to 41,
81, 132, 135, 152 to 153
Verification, 3 to 4, 64
Virtual dimension, 81
VMS, 3, 28 to 29, 32, 63,
73 to 74, 77, 83, 87,
118 to 119, 124, 128,
136 to 137, 140, 142 to 143,
167

WRITE process, 139 to 140

END

FILMED

3-85

DTIC

END

FILMED

3-85

DTIC