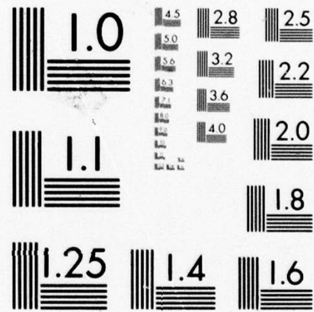
AD-A084 351 NORTHWESTERN UNIV EVANSTON IL DEPT OF ELECTRICAL ENG--ETC F/G 9/2
PERFORMANCE RIPPLE EFFECT ANALYSIS FOR LARGE-SCALE SOFTWARE MAI--ETC(U)
MAR 80 S S YAU, J S COLLOFELLO F30602-76-C-0397

UNCLASSIFIED RADC -TR-80-55 NL



MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

LEVEL II

12

RADC-TR-80-55
Interim Report
March 1980

ADA084351

PERFORMANCE RIPPLE EFFECT ANALYSIS FOR LARGE SCALE SOFTWARE MAINTENANCE

Northwestern University

Stephen S. Yau
James S. Collofello

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED

DTIC
ELECTE
MAY 20 1980
S D
D

ROME AIR DEVELOPMENT CENTER
Air Force Systems Command
Griffiss Air Force Base, New York 13441

DDC FILE COPY

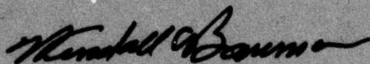
80 5 19 016

This report has been reviewed by the RADC Public Affairs Office (PA) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

RADC-TR-80-55 has been reviewed and is approved for publication.

APPROVED:

ROCCO F. IUORNO
Project Engineer

APPROVED: 

WENDALL C. BAUMAN, Col, USAF
Chief, Information Sciences Division

FOR THE COMMANDER:

JOHN P. HUSS
Acting Chief, Plans Office

If your address has changed or if you wish to be removed from the RADC mailing list, or if the addressee is no longer employed by your organization, please notify RADC (ISIS), Griffiss AFB NY 13441. This will assist us in maintaining a current mailing list.

Do not return this copy. Retain or destroy.

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

Field	Value
1. REPORT NUMBER	RADC-TR-80-55
2. GOVT ACCESSION NO.	AD-A084352
3. RECIPIENT'S CATALOG NUMBER	
4. TITLE (and Subtitle)	PERFORMANCE RIPPLE EFFECT ANALYSIS FOR LARGE-SCALE SOFTWARE MAINTENANCE.
5. TYPE OF REPORT & PERIOD COVERED	Interim Report Nov 77 – Nov 79
6. PERFORMING ORG. REPORT NUMBER	N/A
7. AUTHOR(s)	Stephen S. Yau James S. Collofello
8. CONTRACT OR GRANT NUMBER(s)	F30602-76-C-0397
9. PERFORMING ORGANIZATION NAME AND ADDRESS	Northwester University, Department of Electrical Engineering & Computer Science Evanston IL 60201
10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS	62702F 55810278
11. CONTROLLING OFFICE NAME AND ADDRESS	Rome Air Development Center (ISIS) Griffiss AFB NY 13441
12. REPORT DATE	March 1980
13. NUMBER OF PAGES	125
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office)	Same
15. SECURITY CLASS. (of this report)	UNCLASSIFIED
15a. DECLASSIFICATION/DOWNGRADING SCHEDULE	N/A

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

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

Same

18. SUPPLEMENTARY NOTES

RADC Project Engineer: Rocco F. Iuorno (ISIS)

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

Software maintenance
performance ripple effect analysis
mechanisms for modification propagation
performance attributes
critical sections
maintenance techniques
complexity of program complexity

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

This report analyzes the possible ripple effect of software modifications during the maintenance phase on the performance of the system, and leads to the development of a maintenance technique for predicting which performance requirements in the system may be affected by a proposed modification. This report formalizes the technique outlined in our previous report entitled, "Performance Considerations in the Maintenance Phase of Large-Scale Software Systems". This technique enables maintenance per-

(Cont'd)

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

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

40035

Item 20 (Cont'd)

sonnel to incorporate performance considerations in their criteria for selecting the types and locations of software modifications to be made. After the maintenance changes have been implemented, this technique can also be useful to the retesting effort of the system by identifying which performance requirements must be reverified to insure that they have not been violated by the maintenance activity. This technique is applicable to all types of large-scale software systems possessing performance requirements, including multiprocessing systems.

In the development of this technique, mechanisms for the propagation of performance changes from one part of the system to another are identified. Performance attributes and critical software sections are also identified. The relationship among these mechanisms, performance attributes, critical software sections, and performance requirements forms the basis upon which the maintenance technique analyzes performance ripple effect.

In addition to analyzing the performance ripple effect, the maintenance technique also contributes to the computation of a figure-of-merit for the complexity of a proposed program modification. This figure provides a measure which reflects the amount of work involved in performing maintenance and provides a standard on which comparisons of modifications can be made.

An additional report entitled, "Ripple Effect Analysis for Large-Scale Software Maintenance", constitutes a handbook for applying the ripple effect analysis technique, including for performance ripple effect analysis technique. This handbook describes the procedures to be followed in the application of this technique to large scale programs.

Accession For
NTIS GRA&I ☒
DDC TAB ☐
Unannounced ☐
Justification
By
Distribution/
Availability Codes
Dist. Avail and/or special
A

DTIC
SELECTE
MAY 20 1980
D

CONTENTS

		Page
	List of Figures	iv
1.0	INTRODUCTION	1
1.1	Review of Performance Dependency Relationships	7
1.2	Review of Mechanisms for the Propagation of Performance Changes	7
1.2.1	Parallel Execution	8
1.2.2	Shared Resources	8
1.2.3	Interprocess Communication	8
1.2.4	Called Modules	8
1.2.5	Shared Data Structures	9
1.2.6	Sensitivity to the Rate of Input	9
1.2.7	Execution Priorities	9
1.2.8	Abstractions	10
2.0	PERFORMANCE AND VIRTUAL PERFORMANCE ATTRIBUTES	10
3.0	FORMAL DESCRIPTION OF THE MECHANISMS FOR THE PROPAGATION OF PERFORMANCE CHANGES	15
3.1	Parallel Execution	18
3.1.1	Algorithm to Identify the Parallel Execution Mechanism	25
3.1.2	An Example for Illustrating Algorithm 3.1.1	26
3.1.3	Theorem for Algorithm 3.1.1	30
3.1.4	Identification of Parallel Execution Sets	32
3.1.5	Algorithm to Identify the Parallel Execution Sets in a Program	33
3.1.6	An Example for Illustrating Algorithm 3.1.5	34
3.1.7	Theorem for Algorithm 3.1.5	39
3.2	Shared Resources	41
3.2.1	Algorithm to Identify the Shared Resources Mechanism	43
3.2.2	An Example for Illustrating Algorithm 3.2.1	43
3.2.3	Theorem for Algorithm 3.2.1	47
3.3	Interprocess Communication	48
3.3.1	Algorithm to Identify the Interprocess Communication Mechanism	49
3.3.2	An Example for Illustrating Algorithm 3.3.1	49
3.3.3	Theorem for Algorithm 3.3.1	52
3.4	Called Modules	52
3.4.1	Algorithm to Identify the Called Modules Mechanism	53
3.4.2	An Example to Illustrate Algorithm 3.4.1	53
3.4.3	Theorem for Algorithm 3.4.1	58

		Page
3.5	Shared Data Structure	59
3.5.1	Algorithm to Identify the Shared Data Structures Mechanism	60
3.5.2	Proof of Algorithm 3.5.1	60
3.6	Sensitivity to the Rate of Input	61
3.6.1	Algorithm to Identify the Sensitivity to the Rate of Input Mechanism	62
3.7	Execution Priorities	62
3.7.1	Algorithm to Identify the Execution Priorities Mechanism	63
3.7.2	An Example for Illustrating Algorithm 3.7.1	63
3.7.3	Theorem for Algorithm 3.7.1	63
3.8	Abstractions	66
3.8.1	Algorithm to Identify the Abstraction Mechanism	67
3.8.2	An Example for Illustrating Algorithm 3.8.1	68
3.8.3	Theorem for Algorithm 3.8.1	70
4.0	CRITICAL SECTIONS OF A PROGRAM	71
4.1	Critical Section One	73
4.2	Critical Section Two	73
4.3	Critical Section Three	74
4.4	Critical Section Four	75
4.5	Critical Section Five	76
4.6	Critical Section Six	77
4.7	Critical Section Seven	78
5.0	DEPENDENCY RELATIONSHIPS BETWEEN VIRTUAL PERFORMANCE ATTRIBUTES AND PERFORMANCE ATTRIBUTES	79
5.1	Dependency Relationships for Virtual Performance Attribute of Type One	79
5.2	Dependency Relationships for Virtual Performance Attribute of Type Two	80
5.3	Dependency Relationships for Virtual Performance Attribute of Type Three	81
5.4	Dependency Relationships for Virtual Performance Attribute of Type Four	82
5.5	Dependency Relationships for Virtual Performance Attribute of Type Five	83
6.0	PERFORMANCE DEPENDENCY RELATIONSHIP RULES	84
7.0	RIPPLE EFFECT OF PERFORMANCE CHANGE	88

Page

8.0 MAINTENANCE TECHNIQUE FOR PREDICTING WHICH PERFORMANCE REQUIREMENTS ARE AFFECTED BY THE MAINTENANCE ACTIVITY 90

8.1 Key Algorithms and Guidelines Composing the Maintenance Technique 91

8.1.1 Guidelines for the Decomposition of Performance Requirements into Performance Attributes 91

8.1.2 Algorithm to Identify All of the Mechanisms for the Propagation of Performance Changes Present in the Program . . . 93

8.1.3 Algorithm to Identify All of the Critical Sections in the Program 95

8.1.4 Algorithm to Identify All of the Performance Attributes in the Program 96

8.1.5 Algorithm to Identify All of the Virtual Performance Attributes in a Program 97

8.2 Formal Description of the Maintenance Technique 98

8.2.1 Lexical Analysis Stage of the Maintenance Technique 98

8.2.2 Algorithm to Trace the Ripple Effect of Performance Changes Among the Performance Attributes 99

8.2.3 Tracing Stage of the Maintenance Technique 100

8.3 Proof of Algorithms Composing the Maintenance Technique 101

8.3.1 Proof of the Algorithms Composing Stage One of the Maintenance Technique 101

8.3.2 Theorem for Algorithm 8.2.2 103

8.3.3 Proof of the Second Stage of the Maintenance Technique 104

8.4 Application of the Maintenance Technique to the Retesting Phase of the Maintenance Process 105

9.0 FIGURE OF MERIT FOR THE COMPLEXITY OF A PROGRAM MODIFICATION . . 106

10.0 FUTURE RESEARCH AND CONCLUSION 109

10.1 Dynamic Analysis 109

10.2 Refined Estimator of the Complexity of Program Modification from a Performance Perspective 110

10.3 Application to Design Phase 112

10.4 Conclusion 113

11.0 REFERENCES 113

List of Figures

		Page
Figure 1.	A survey of large-scale program life cycle cost [2]	2
Figure 2.	A more recent survey of large-scale program life cycle cost [5]	3
Figure 3.	An example showing the relationship of performance attributes and the mechanisms for propagation of performance changes	11
Figure 4.	An illustration of the relationship of virtual performance attributes to performance attributes of a program	16
Figure 5.	The R-Net of a sample program	19
Figure 6.	The R-Net of a more complex program	21
Figure 7.	The R-Net of another sample program	23
Figure 8.	The refined R-Net of a sample program	24
Figure 9.	The refined R-Net of a program for illustrating Algorithm 3.1.1	27
Figure 10.	An example for illustrating Algorithm 3.1.5	35
Figure 11.	An example for illustrating Algorithm 3.2.1	44
Figure 12.	An example for illustrating Algorithm 3.3.1	50
Figure 13.	An example for illustrating Algorithm 3.4.1	54
Figure 14.	An example for illustrating Algorithm 3.7.1	64
Figure 15.	An example for illustrating Algorithm 3.8.1	69
Figure 16.	An example for illustrating a possible relationship among performance attributes (P.A.), critical sections (C.S.) and the mechanisms for propagation of performance changes in a program	72
Figure 17.	An example to illustrate the ripple effect of performance changes	89

Page

Figure 18. An example for illustrating a possible relationship among performance requirements (P.R.), performance attributes (P.A.), critical sections (C.S.) and the mechanisms for propagation of performance changes in a program 94

Figure 19. The framework of the performance ripple effect analysis technique for predicting which performance requirements are affected by the maintenance activity . . . 102

Figure 20. The performance ripple effect analysis technique as a part of the maintenance process 107

Figure 21. A directed graph representing resource utilization of a process 111

1.0 INTRODUCTION

The major expenses in computer systems at present and in the future are in software [1,2]. While the cost of hardware is dropping rapidly due to an order-of-magnitude improvement in the hardware price/performance ratio every ten years, software productivity improves only slowly. Thus, the cost of software relative to hardware is increasing. Estimates of this overall cost of software development and maintenance in the United States range from $15 to $25 billion. By 1985, computer software expenses are estimated to be about 90 percent of the total system cost [1,2]. In Fact, on one existing program, the World Wide Military Command and Control System, this percent is already in that vicinity with $722 million for software to $50-$100 million for hardware [3].

These staggering figures have stimulated great efforts in recent years to study software. Yet, the cost of developing large-scale software systems has become unacceptably high. The term large-scale software as used in this report refers to those programs in excess of 30,000 lines of code requiring a team effort to produce and at least one year to implement. Such systems are becoming increasingly important in a variety of applications including military weapon systems, vehicle control systems, industrial operations and process control systems, communication systems, and the like. Because these systems are inherently complex, they require a great deal of time, effort, and cost to develop.

Current experience in developing these large-scale software systems has been discouraging. Most software development projects are unsuccessful in terms of specification, time, and cost [1]. An examination of the software's life cycle was conducted by Boehm [2] to analyze the software phases. The results of his study are illustrated in Figure 1. His results indicate that the cost of maintenance is 40 percent of the total software cost. More recent studies have confirmed the high cost of maintenance [4-7]. Figure 2 is a more recent analysis of the software phases [5]. The results of this analysis indicate that the cost of maintenance is 67 percent of the total software cost. Other studies have reported similar figures for the cost of maintenance. For example, it has been estimated that up to 80 percent of IBM's application development resources are devoted to maintenance [6].

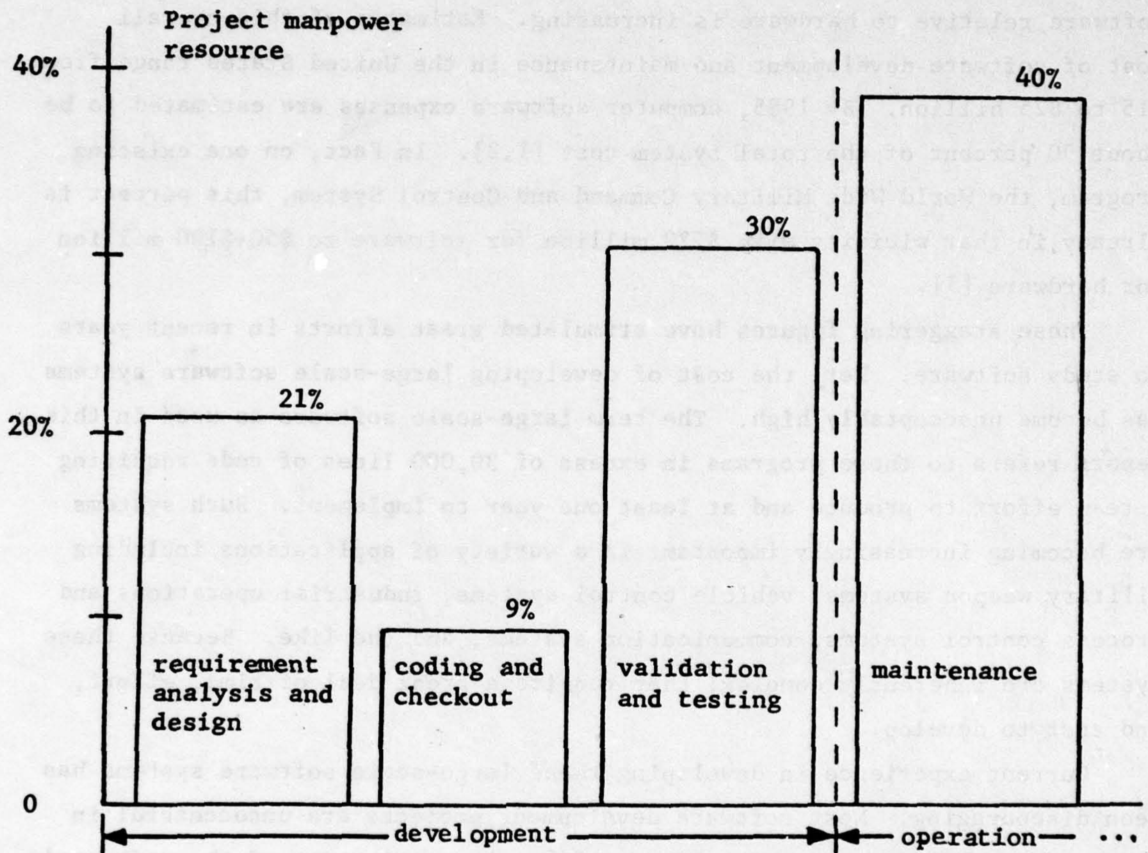


Figure 1. A survey of large-scale program life cycle cost [2].

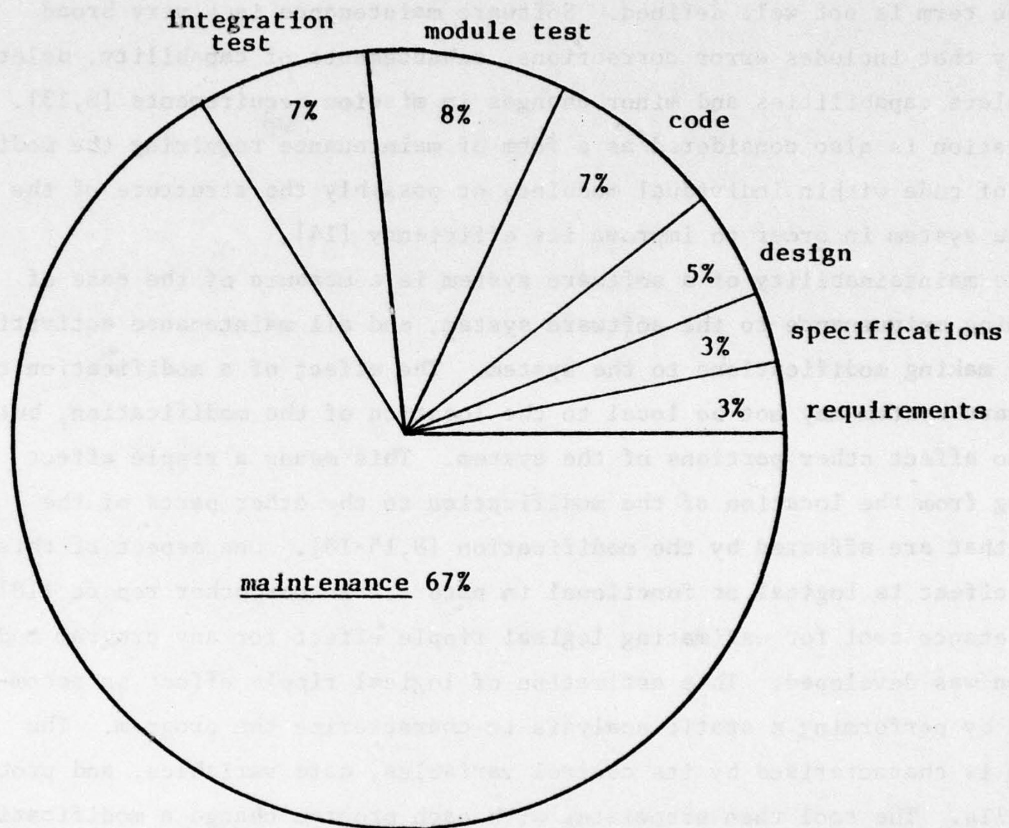


Figure 2. A more recent survey of large-scale program life cycle cost [5].

Similar figures were also reported by Elshoff [7], who indicated that 75 percent of General Motor's commercial software effort was spent on maintenance. With maintenance costs so high, it would appear that this phase should be analyzed in an effort to help reduce the total software cost.

When one attempts to discuss software maintenance, it becomes apparent that the term is not well defined. Software maintenance is a very broad activity that includes error corrections, enhancements of capability, deletion of obsolete capabilities and minor changes in mission requirements [8,13]. Optimization is also considered as a form of maintenance requiring the modification of code within individual modules, or possibly the structure of the complete system in order to improve its efficiency [14].

The maintainability of a software system is a measure of the ease of performing maintenance to the software system, and all maintenance activities require making modifications to the system. The effect of a modification to a software system may not be local to the location of the modification, but may also affect other portions of the system. This means a ripple effect existing from the location of the modification to the other parts of the system that are affected by the modification [8,15-18]. One aspect of this ripple effect is logical or functional in nature. In our other report [18], a maintenance tool for estimating logical ripple effect for any program modification was developed. This estimation of logical ripple effect is accomplished by performing a static analysis to characterize the program. The program is characterized by its control variables, data variables, and procedure calls. The tool then associates with each program change a modification to a group of decisions which characterizes a portion of the program's specification. Modification to a group of decisions requires physical changes to module variables which must reflect those decisions. A change to a variable defines a primary source from which inconsistency propagates to other program areas. By knowing the primary sources involved in a program modification and the module's characteristics, potential worst logical ripple effect can be calculated by tracing the program paths over which inconsistencies can flow. The development of this tool enables maintenance personnel to do a better job by allowing them to know the scope of effect of a software modification.

Another aspect of this ripple effect, which cannot be handled by the logical ripple effect tool just described, concerns the performance of the system [8,17]. During software maintenance, it is possible to perform a modification to the system, investigate its logical ripple effect and locate the inconsistencies introduced throughout the program by the modification. After all the logical corrections have been made to the system, the maintenance personnel may conclude that they have restored the system to its previous level of functional correctness. The performance of the system, however, may have been altered as a direct result of this maintenance activity. Since a large-scale program usually has both functional and performance requirements, the net result of the maintenance effort may be satisfactory to the functional requirements, but not to some performance requirements [8,17].

In many large-scale systems, the violation of a performance requirement is equivalent to a system error and, thus, requires further corrective action [19,20]. Consequently, it is important in the maintenance process to fully understand the potential effect of a modification to the system in terms of the performance of the parts of the system directly involved in the modification as well as those that are affected indirectly. The change in performance of these parts may then have an impact on the performance of the other parts of the system.

The maintenance process can, thus, be improved if maintenance personnel are supplied with information enabling them to incorporate performance considerations in their criteria for selecting the type and location of software modifications to be made. This information is provided by the development of a maintenance technique for predicting which performance requirements in the system may be affected by a proposed modification. The prediction of which performance requirements may be affected by a software modification is a difficult task. Due to the size and complexity of design of many large-scale programs, maintenance changes can cause repercussions almost anywhere throughout the system [7,8,17]. Thus, the significance of this technique lies in its ability to trace these repercussions and predict which performance requirements may be affected.

In this report we formalize the technique outlined in our previous reports [8,17] for predicting which performance requirements in the system

may be affected by a proposed software modification. This technique is applicable to all types of large-scale systems possessing performance requirements including multiprocessing systems.

In this report we will also illustrate how the technique can help retest the system whether its performance requirements have been violated by the maintenance effort after the maintenance changes have been implemented. The maintenance technique analyzes the proposed changes with respect to the performance of the entire system, and not just the local areas involved in the modification. It can then determine the performance requirements affected by the maintenance change. It should be noted that during the evaluation of alternative modifications, the maintenance technique was used to predict worst-case effects on performance of software modifications. Now that the modification has been completely implemented, the maintenance technique can substantially refine its analysis and determine more accurately the performance requirements affected by the maintenance modifications. This permits the identification of which portions of the system must be retested to insure that these performance requirements have not been violated. Since the maintenance effort must not violate any functional or performance requirements, this maintenance technique provides a significant contribution in determining the scale of retesting effort needed to insure that these requirements have not been violated.

Another very significant product of the analysis of ripple effect from both a logical and performance perspective is the computation of a figure-of-merit for the complexity of a proposed program modification. The figure presented in this report provides a measure which reflects the work involved in performing maintenance and, thus, provides a standard for comparison the complexity of proposed modifications.

It should be noted that the maintenance technique presented in this report is language independent and applicable to currently existing programs as well as newly implemented programs incorporating state-of-the-art design techniques. The technique does not provide maintenance personnel with proposals for modifying the program. Instead, the technique is applied after the maintenance personnel have generated a number of possible maintenance proposals. A figure-of-merit for the complexity of modification can be computed for each of the proposed modifications and the maintenance

personnel can then select the best modification from both a logical and performance perspective. The modification can then be completely implemented, and the changes felt in other portions of the program will be dealt with by analysis of the ripple effect.

1.1 Review of Performance Dependency Relationships

The identification of modules whose performance may change as a consequence of software modifications is a complex task. The identification is complicated by the fact that performance dependencies often exist among modules which are otherwise functionally and logically independent. In our previous report [17], we defined several types of performance dependency relationships and provided examples of each. A brief summary of these definitions is given here.

Performance Dependency Relationship (PDR): A PDR is defined to exist from Module A to Module B if there exists a change in Module A that can have an effect on the performance of Module B.

Performance Interdependency Relationship (PIR): A PIR is defined to exist between Module A and Module B if a PDR exists from Module A to Module B and from Module B to Module A.

Pure Performance Dependency Relationship (PPDR): A PPDR is defined to exist from Module A to Module B if there exists a change in performance of Module A which is not the result of a modification to Module A and can have an effect on the performance of Module B.

Pure Performance Interdependency Relationship (PPIR): A PPIR is defined to exist between Module A and Module B if a PPDR exists from Module A to Module B and from Module B to Module A.

Performance Independent (PI): Two modules are defined to be performance independent if there does not exist a PDR or PPDR from one to the other.

1.2 Review of Mechanisms for the Propagation of Performance Changes

It is obvious that when a logical or functional error is discovered in the software, the scope of effect of this error will include other modules. Analogously, when a performance change is made, the scope of effect of the change can be determined by examining the mechanisms by which this change

can affect other modules. In our previous report [17], we identified eight mechanisms which may exist in large-scale systems by which changes in performance as a consequence of a software modification are propagated throughout the system. These mechanisms are summarized here:

1.2.1 Parallel Execution

The first mechanism for the propagation of performance changes involves a modification during maintenance which results in a loss of parallel execution capability. In the maintenance phase it is possible to introduce software modifications to a module which can destroy its ability to be executed with other modules in parallel. The maintenance personnel must then be aware of the change in performance which will result from the loss of the parallelism. Major changes in performance may occur due to execution delays and contention for resources previously alleviated through the parallel execution.

1.2.2 Shared Resources

Another mechanism for the propagation of performance changes is contention for resources among modules. When modules are forced to share resources, the time when each module requests and releases common resources are important performance parameters. Thus, software modifications producing performance changes in the time that the resources are utilized can have detrimental effects on the performance of modules that must share the resources.

1.2.3 Interprocess Communication

Another mechanism for the propagation of performance changes involves communication among the modules in the system. When one module must send a message to another module, the performance of the module receiving the message is dependent of when the message is actually received. Thus, modifications to the module sending the message that alter the time when the message is sent can affect the performance of the module designated to receive the message.

1.2.4 Called Modules

Another mechanism for the propagation of performance changes is the utilization of called modules in the software. Modifications to modules in the maintenance phase can be divided into two types. A <u>bounded modification</u>

to a module is a modification which does not alter the performance of the module. An unbounded modification to a module is a modification which alters the performance of the module. An unbounded modification to the called module will affect the performance of all modules calling it. A bounded modification will not affect the performance of the other modules calling it.

1.2.5 Shared Data Structures

Another mechanism for the propagation of performance changes is through the utilization of shared data structures. The basic dynamic attributes contributing to performance in this area are a module's storage and retrieval times for entries in the data structures. Factors affecting storage and retrieval times vary among the different types of data structures as well as the algorithms utilized to manipulate these structures.

1.2.6 Sensitivity to the Rate of Input

Another mechanism for the propagation of performance changes involves the rate of input into a process. This change in input frequency is the result of a changing environment. The resulting change of input rate to the process can have major repercussions in terms of its functional and performance requirements. For example, it can lead to saturation and possibly overflow of data structures involved with the processing of the input. The increased frequency of input arrivals may also lead to interruptions in processing which can lead to both functional and performance requirement violations.

1.2.7 Execution Priorities

Another mechanism for the propagation of performance changes involves the execution priority of modules. During the development phase, module execution priorities may have been established to insure correct sequencing or preservation of critical system performance requirements. The priorities are used to determine the execution order of modules capable of beginning execution at the same time. The priorities may also be utilized in the determination of whether or not a module's execution should be pre-empted for that of a module with a higher priority. During the maintenance phase, it is important for maintenance personnel to recognize the effect of a proposed modification in respect to the existing priorities in the system.

1.2.8 Abstractions

Another mechanism for the propagation of performance changes is the utilization of abstractions in the software. The use of abstractions is a popular design tool and adds to the maintainability of the system by hiding design decisions. From the performance perspective of maintainability, however, abstractions are "trojan horses." This is because a change in the implementation of the abstraction will very likely affect the performance of the abstraction, and, thus, the performance of all modules utilizing the abstraction. This is a classic example of a case where a modification to the software during maintenance does not produce any functional or logical changes, but it does result in performance changes.

2.0 PERFORMANCE AND VIRTUAL PERFORMANCE ATTRIBUTES

Performance attributes of a program are defined as attributes corresponding to measurements of key portions of the execution of the program. For example, one performance attribute of a module is its execution time. Another is the utilization for a particular resource during the execution of the program. There is a distinct relationship between performance attributes and the eight mechanisms for the propagation of performance changes. The eight mechanisms operate as links between performance attributes of modules. In other words, a change in a performance attribute of one module can affect a performance attribute in another module via one of the eight mechanisms. For example, let X represent the performance attribute corresponding to the time a resource is seized by module A. Assume module B is in contention for the same resource with module A and let Y represent the performance attribute corresponding to the time module B seizes the same resource. Then a change in performance attribute X can affect performance attribute Y via the shared resources mechanism.

The relationship between performance attributes and the eight mechanisms for the propagation of performance changes is illustrated in Figure 3, where the directed line labeled with a mechanism connecting two performance attributes indicates a performance dependency relationship exists between the performance attributes. For example, if performance attribute 2 of Process A is modified, it can affect performance attribute 2 of Process B

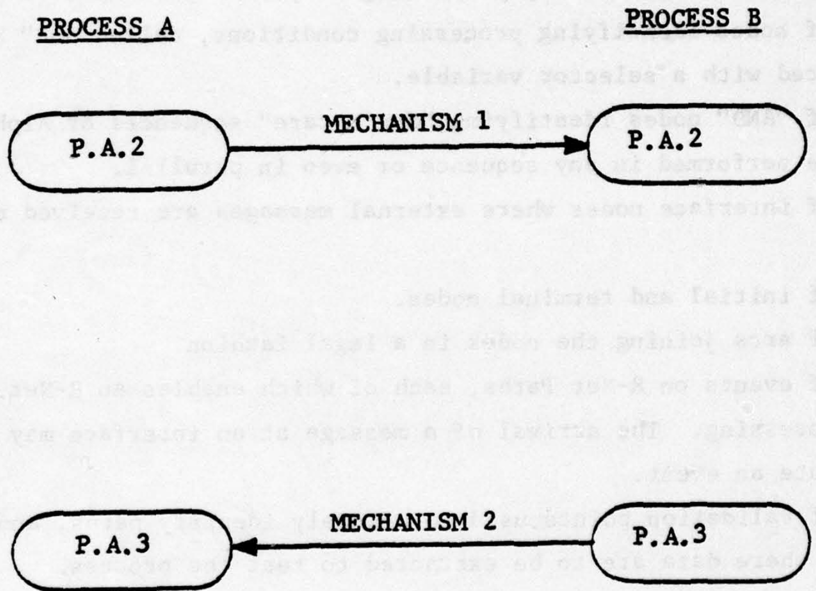


Figure 3. An example showing the relationship of performance attributes and the mechanisms for propagation of performance changes.

via mechanism 1.

We will now present fourteen software performance attributes. These performance attributes are not a complete set of attributes corresponding to measurements of the execution of the program. Instead, these performance attributes are the attributes linked with the eight mechanisms as previously discussed. One way to state the performance requirements of software is to express them in terms of flows through the system, such as in an R-Net the performance attributes being defined for "alphas" [17,21]. We may define an R-Net as a graph that consists of the following items:

(1) A set of nodes representing processing steps called "alphas".

(2) A set of nodes identifying processing conditions, called "OR" nodes, associated with a selector variable.

(3) A set of "AND" nodes identifying "don't-care" sequences of Alphas which could be performed in any sequence or even in parallel.

(4) A set of interface nodes where external messages are received or transmitted.

(5) A set of initial and terminal nodes.

(6) A set of arcs joining the nodes in a legal fashion.

(7) A set of events on R-Net Paths, each of which enables an R-Net to perform processing. The arrival of a message at an interface may also constitute an event.

(8) A set of validation points used to uniquely identify paths, and to specify where data are to be extracted to test the process.

The R-Net is a synthesis of a set of paths. Each Alpha is associated with two subsets of data identifiers, representing data input to and output from the processing step. The link between the functional and performance requirements is achieved through the validation points. Each validation point is associated with a set of data identifiers representing data to be collected.

The following eleven performance attributes are defined for "alphas" in an R-Net which correspond to modules. For a given module, not all of these performance attributes may be applicable.

Performance Attribute 1: The ability of the module being executed in parallel with another module.

Performance Attribute 2: For each resource in contention, the relative time that the module seizes the resource.

Performance Attribute 3: For each resource in contention, the relative time that the module releases the resource.

Performance Attribute 4: The relative time that the module begins execution.

Performance Attribute 5: The relative time that the module transmits a message to another module.

Performance Attribute 6: The execution time of the module.

Performance Attribute 7: For each resource utilized in the module, the resource utilization by the module. Resource utilization is defined as the time that the module possesses the resource.

Performance Attribute 8: For each data structure, the storage and retrieval times for entries in the data structure.

Performance Attribute 9: For each data structure, the number of entries in the data structure.

Performance Attribute 10: For each data structure, the service time of an entry in the data structure, i.e., the relative time that an entry remains in the data structure before being serviced.

Performance Attribute 11: The rate of input to the module.

Two additional performance attributes can be defined for "alphas" in an R-Net which do not correspond to modules. These performance attributes are needed since some "alphas" in the R-Net are critical to the determination of whether or not certain performance requirements are affected by program modification. These critical "alphas" will be discussed in detail in Section 8.1.1.

Performance Attribute 6': The execution time of an "alpha" contained in a module.

Performance Attribute 7': The resource utilization of an "alpha" contained in a module for each resource utilized.

An additional performance attribute can also be defined for a module. This performance attribute is not related to the eight mechanisms for the propagation of performance changes as are the other thirteen performance attributes. Thus, this performance attribute is not affected by a change to a performance attribute in another module. It does, however, correspond to a measurement of a key portion of the execution of the program. The following is the additional performance attribute.

Performance Attribute 12: For each dependent iterative structure in the module, the number of iterations. A dependent iterative structure is an iterative structure which does not possess a constant number of iterations, i.e., it has a variable number of iterations depending upon certain program variables.

These fourteen software performance attributes form the basis of performance ripple effect. All performance changes in the program will be analyzed in terms of these performance attributes. When a performance attribute is affected by a modification, the other performance attributes affected by the change must be identified. It is not, however, always possible to identify exactly which performance attributes are affected as a consequence of the performance ripple effect. When a particular performance attribute is affected by a modification, it is possible to identify the critical sections of code which may experience the performance change. Corresponding to each critical section there may be several performance attributes. The concept of a virtual performance attribute is introduced to represent this change in performance of a critical section.

A virtual performance attribute is defined to represent a change in performance of a critical section which is a consequence of affecting some performance attribute in the program. If a performance attribute is involved in a performance dependency relationship with a virtual performance attribute, it means that a change in the performance attribute will affect the virtual performance attribute, i.e., the performance of some software section. Corresponding to the virtual performance attribute, there may be many performance attributes.

The virtual performance attributes of a program are the following:

Virtual Performance Attribute 1: The execution time of the critical sections of a competing module which must wait due to denial of a particular contended resource.

Virtual Performance Attribute 2: The execution time of the critical sections of a module which contain an invocation to a module or the utilization of an abstraction.

Virtual Performance Attribute 3: The execution time of the critical sections of a module which contain a dependent iterative structure.

Virtual Performance Attribute 4: The execution time of the critical sections of code which contain a storage or retrieval request of an entry in a data structure.

Virtual Performance Attribute 5: The execution time of the critical sections of code which must wait for a message to be transmitted from another module.

In Section 5.0, algorithms will be presented for identifying the dependency relationships between virtual performance attributes and performance attributes. These relationships are a key in tracing the performance ripple effect among performance attributes. For example, if we affect performance attribute 6 of Module A of Program X in Figure 4, we will affect virtual performance attribute 2 of Program X. Now corresponding to virtual performance attribute 2 of Program X is performance attribute 5 of Program X. Thus, the actual effect of changing performance attribute 6 of module A is a change in performance attribute 5 of Program X via virtual performance attribute 2.

3.0 FORMAL DESCRIPTION OF THE MECHANISMS FOR THE PROPAGATION OF PERFORMANCE CHANGES

In this section, we will provide a formal description of the eight mechanisms for the propagation of performance changes summarized in Section 1.2. The relationship of these mechanisms, performance attributes, and critical sections will be shown to form the basis for the concept of a performance change ripple-effect as a consequence of software modifications.

PROGRAM X

START

.

.

.

CALL MODULE A

.

.

.

SEND MESSAGE 1

.

.

.

STOP

Figure 4. An illustration of the relationship of virtual performance attributes to performance attributes of a program.

When a critical section is modified, it may affect the corresponding performance attributes. A change in these performance attributes may then ripple, i.e. affect other performance attributes via any applicable mechanisms. These performance attributes may then, in turn, affect still other performance attributes via the applicable mechanisms.

A performance dependency relationship is defined to exist between two performance attributes if a change in one of the performance attributes may affect the other performance attribute. If the performance attributes are for different modules, then a change in one of the performance attributes may affect the other performance attribute only via one of the mechanisms for the propagation of performance changes.

In this section, we will identify these performance dependency relationships among the performance attributes in the program for each of the mechanisms for the propagation of performance changes. The performance dependency relationships among the performance attributes in the program are described according to a set of rules.

The rules are of the format

MODULE X/PA.Y → MODULE Z/PA.W. CONDITION

and are interpreted as follows:

> A change in performance attribute Y of module X may affect performance attribute W of module Z if the condition is satisfied.
>
> *A variation of this format* is the replacement of MODULE with DS. which represents data structures. DS. X/PA.Y. is then interpreted as the Yth performance attribute of data structure X.

In this section, we will also identify the performance dependency relationships in existence between the performance attributes and the virtual performance attributes. The performance dependency relationships between the performance attributes and virtual performance attributes in the program are described according to a set of rules.

The rules are of the format

MODULE X/PA.Y → MODULE Z/VPA.W. CONDITION

and are interpreted as follows:

A change in performance attribute Y of module X may affect virtual performance attribute W of module Z if the condition is satisfied.

In this section, we will also formally describe algorithms for identifying each of the mechanisms for the propagation of performance changes in the program. Also included in this section will be illustrative examples of the algorithms and proofs that the algorithms are correct.

3.1 Parallel Execution

The first mechanism for the propagation of performance changes to be formally described is the parallel execution mechanism. This mechanism analyzes the performance changes resulting from a loss of parallel execution capability. The major performance change will be experienced in the process in which the module was executed in parallel. The primary effect will be an increase in execution time as a consequence of the lost parallelism. The delay can be considerable if the modified module must wait for resources that were previously available at its earlier execution time.

Performance attribute 1 is associated with this mechanism. The parallel execution mechanism can be defined in terms of the following performance dependency rules:

MODULE X/PA.1 → MODULE Y/PA.4 if X is involved in a PIR with module Y via mechanism 1.

MODULE X/PA.1 → MODULE Y/PA.6 if X is in a PIR with Y via mechanism 1.

An algorithm has been developed for identifying the existence of the parallel execution mechanism in a program and the modules in the program affected by this mechanism. This algorithm identifies all of the modules executable in parallel in the program. The determination of which modules can be executed in parallel is a decision made during the design phase of the system. This decision must be reflected in either the software implementation or its accompanying documentation. In either case, it can be illustrated through the use of R-Nets [17].

The identification of which modules may be executed in parallel based upon the information in an R-Net is easy in a small program. For example, in the program illustrated in Figure 5, it is obvious that modules B and C,

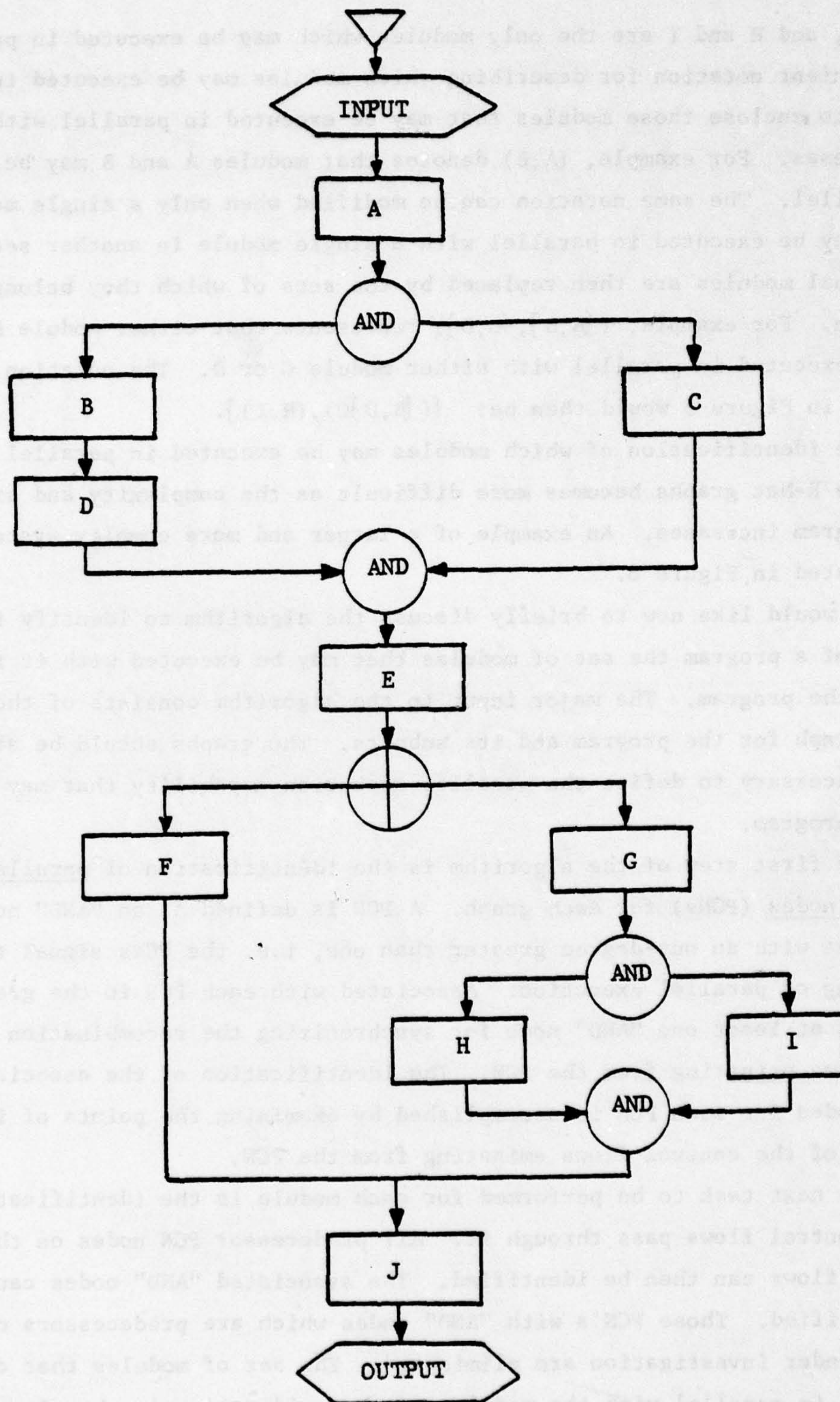


Figure 5. The R-Net of a sample program.

C and D, and H and I are the only modules which may be executed in parallel. A convenient notation for describing which modules may be executed in parallel is to enclose those modules that may be executed in parallel within parentheses. For example, (A,B) denotes that modules A and B may be executed in parallel. The same notation can be modified when only a single module in a set may be executed in parallel with a single module in another set. The individual modules are then replaced by the sets of which they belong in the notation. For example, ({A,B},{C,D}) represents that either module A or B may be executed in parallel with either module C or D. The notation for the example in Figure 5 would then be: {({B,D}C),(H,I)}.

The identification of which modules may be executed in parallel based upon the R-Net graphs becomes more difficult as the complexity and size of the program increases. An example of a larger and more complex system is illustrated in Figure 6.

We would like now to briefly discuss the algorithm to identify for each module of a program the set of modules that may be executed with it in parallel in the program. The major input to the algorithm consists of the main R-Net graph for the program and its subnets. The graphs should be at the level necessary to define the parallel execution capability that may exist in the program.

The first step of the algorithm is the identification of parallel control nodes (PCNs) for each graph. A PCN is defined as an "AND" node in the R-Net with an out-degree greater than one, i.e. the PCNs signal the beginning of parallel execution. Associated with each PCN in the graph, there is at least one "AND" node for synchronizing the recombination of control flows eminating from the PCN. The identification of the associated "AND" nodes for each PCN is accomplished by examining the points of intersection of the control flows eminating from the PCN.

The next task to be performed for each module is the identification of which control flows pass through it. All predecessor PCN nodes on these control flows can then be identified. The associated "AND" nodes can then be identified. Those PCN's with "AND" nodes which are predecessors of the module under investigation are eliminated. The set of modules that can be executed in parallel with the module under consideration is then formed by adding to the set all modules on control flows eminating from the remaining

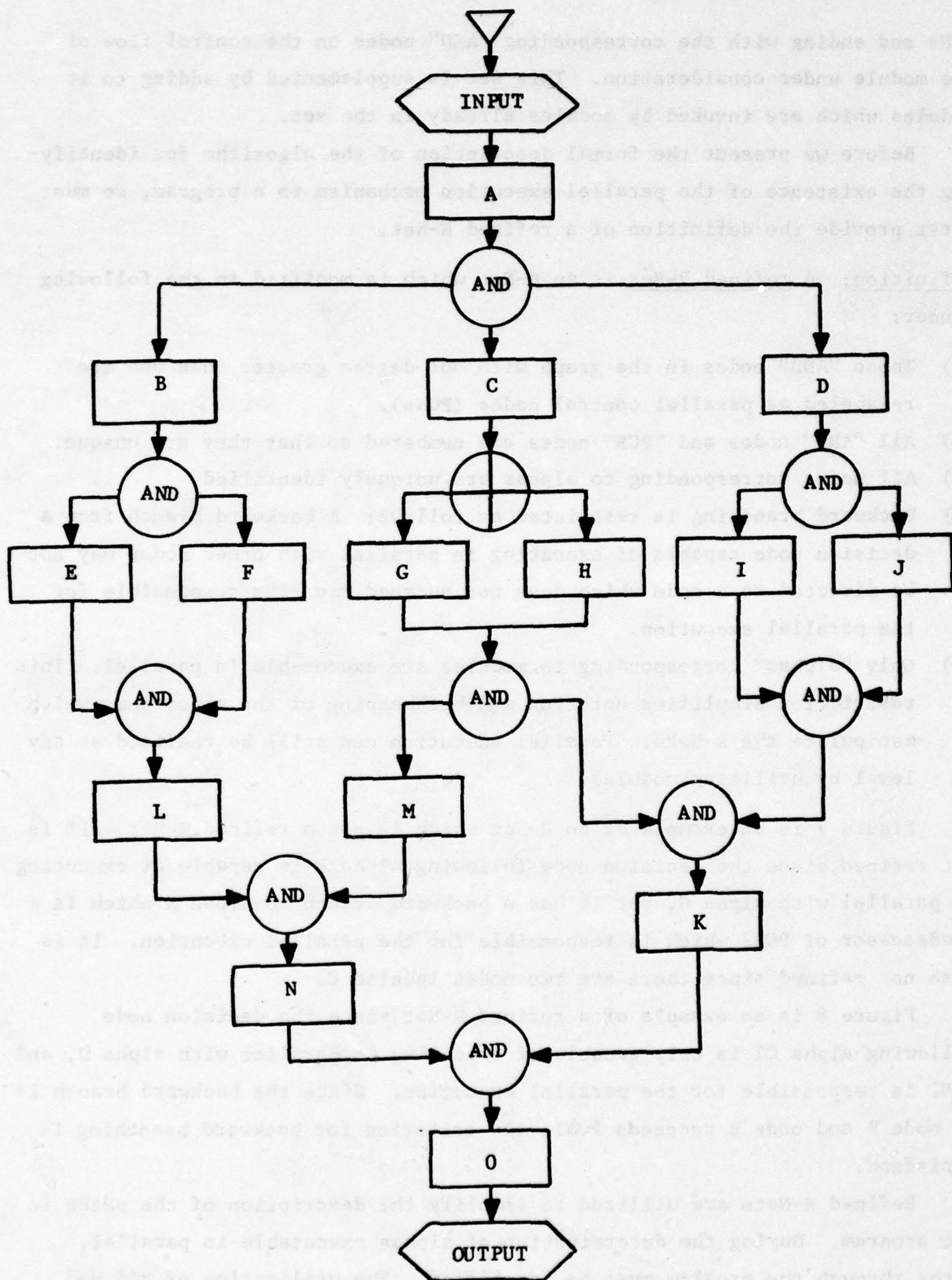


Figure 6. The R-Net of a more complex program.

PCNs and ending with the corresponding "AND" nodes on the control flow of the module under consideration. This set is supplemented by adding to it modules which are invoked by modules already in the set.

Before we present the formal description of the algorithm for identifying the existence of the parallel execution mechanism in a program, we must first provide the definition of a refined R-Net.

Definition: A refined R-Net is an R-Net which is modified in the following manner:

(1) Those "AND" nodes in the graph with out-degree greater than one are relabeled as parallel control nodes (PCNs).

(2) All "AND" nodes and "PCN" nodes are numbered so that they are unique.

(3) All nodes corresponding to alphas are uniquely identified.

(4) Backward branching is restricted as follows: A backward branch from a decision node capable of executing in parallel with other nodes may not be directed to a node which does not succeed the PCNs responsible for the parallel execution.

(5) Only "alphas" corresponding to modules are executable in parallel. This restriction simplifies notation and bookkeeping of the algorithms which manipulate the R-Nets. Parallel execution can still be realized at any level by utilizing modules.

Figure 7 is an example of an R-Net which is not a refined R-Net. It is not refined since the decision node following alpha F is capable of executing in parallel with alpha G, yet it has a backward branch to alpha B which is a predecessor of PCN2 which is responsible for the parallel execution. It is also not refined since there are two nodes labeled C.

Figure 8 is an example of a refined R-Net since the decision node following alpha C2 is only capable of executing in parallel with alpha D, and PCN1 is responsible for the parallel execution. Since the backward branch is to node B and node B succeeds PCN1, the criterion for backward branching is satisfied.

Refined R-Nets are utilized to simplify the description of the paths in the program. During the determination of alphas executable in parallel, paths through the program must be identified. The utilization of refined R-Nets enables backward branches to be ignored in path identification since

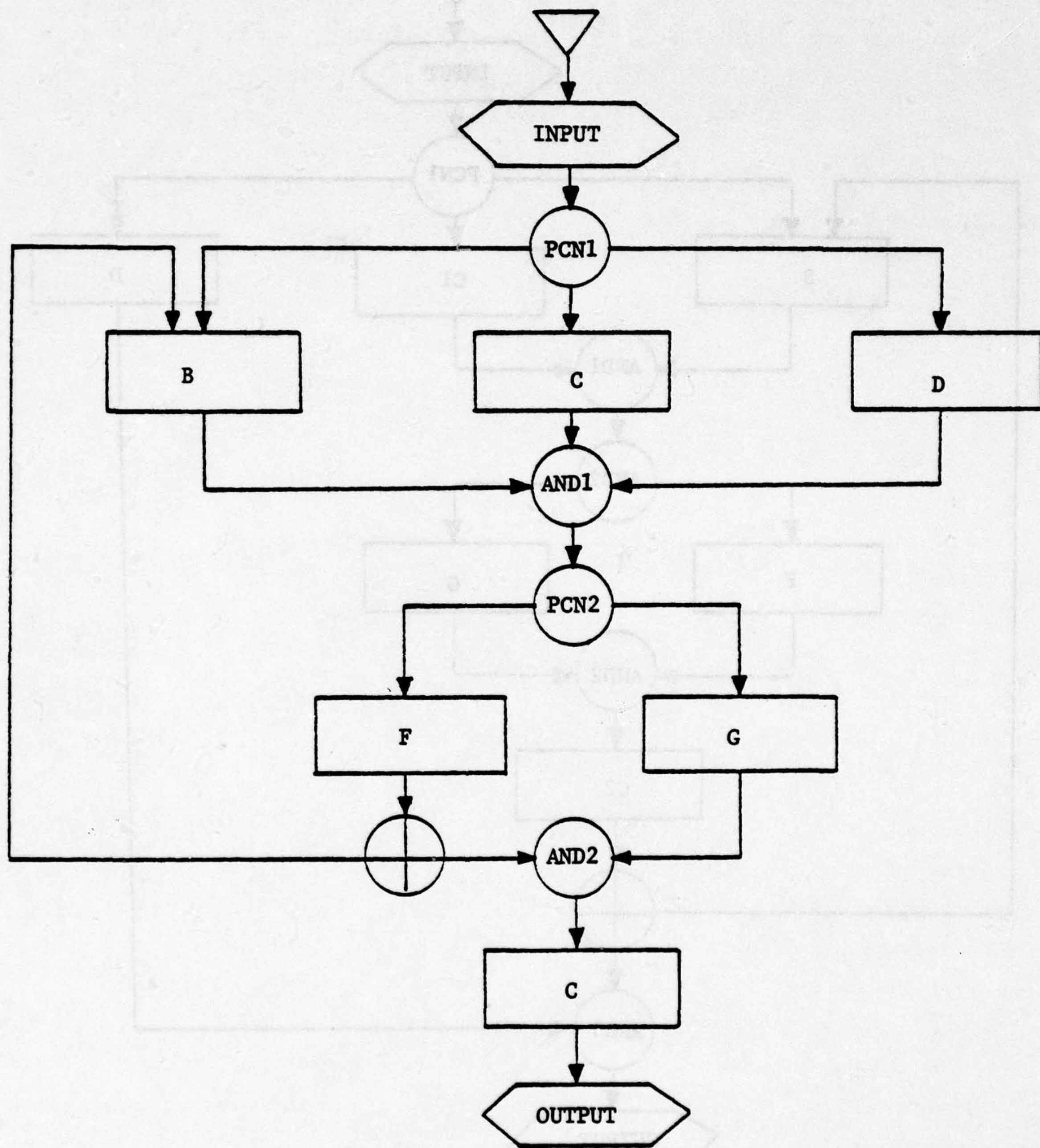


Figure 7. The R-Net of another sample program.

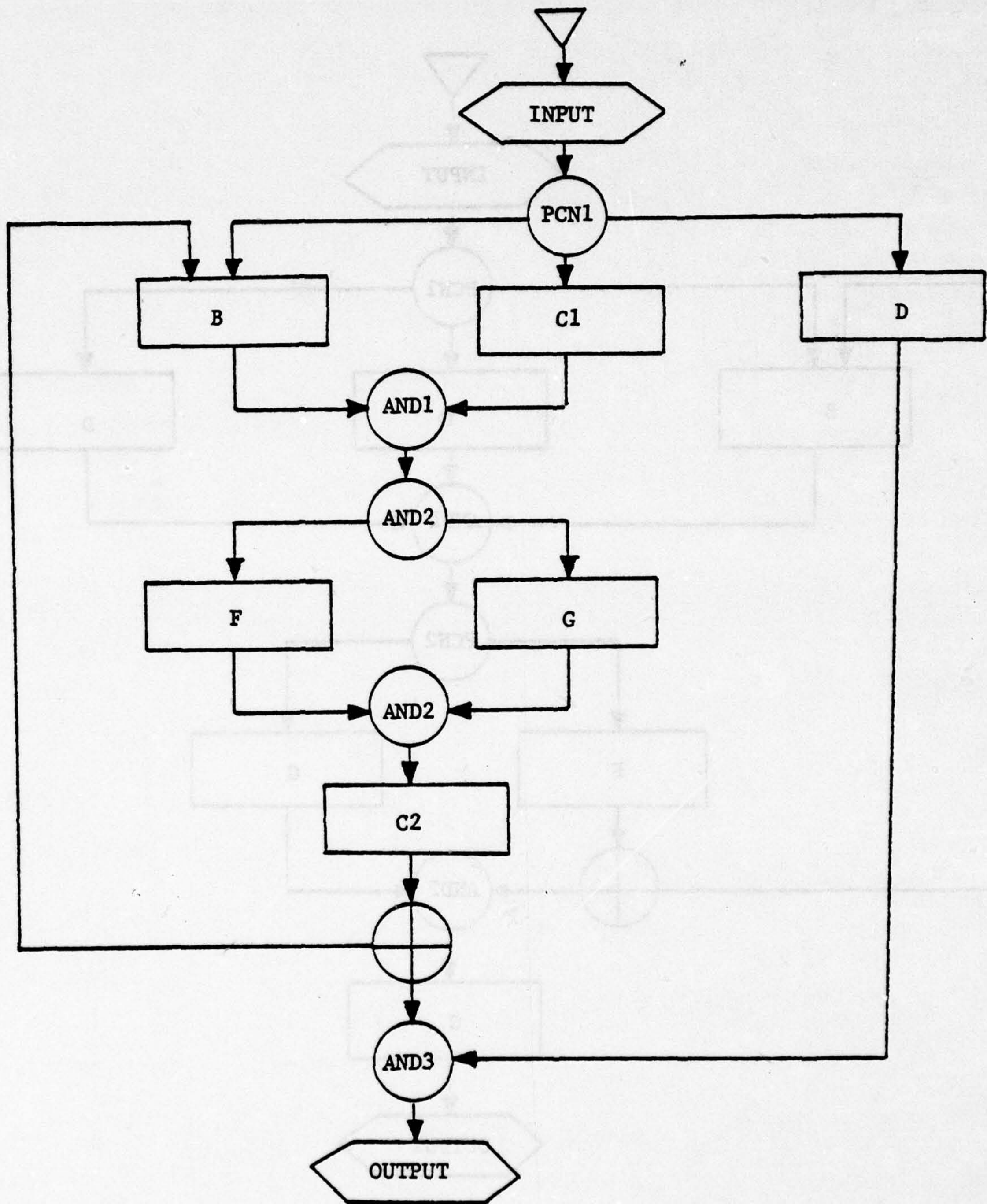


Figure 8. The refined R-Net of a sample program.

the additional paths generated by the backward branches do not contain any new alphas that can be executed in parallel and have not been already identified on the paths without the backward branches.

It should be noted that the refined R-Net should be at a level of abstraction necessary to define the parallel execution in the program. Each alpha in the R-Net corresponds to either a module or a segment of code. For each alpha that corresponds to a module, there exists a subnet for the module.

3.1.1 Algorithm to Identify the Parallel Execution Mechanism

Algorithm 3.1.1

Step 1: Initialize $h_{\hat{n}} = \emptyset$ for each alpha $\hat{n}$ which corresponds to a subnet in the graph. $h_{\hat{n}}$ is defined to contain the set of alphas capable of being executed in parallel with alpha $\hat{n}$. Perform Steps 2-7 for the main program R-Net and each of the subnets containing alphas executable in parallel.

Step 2: Identify all PCNs in the R-Net under consideration. Let $P = \{$all PCNs$\}$.

Step 3: For each PCN i in P and each branch j from PCN i, define the sets $S_{ijk} = \{$nodes in the graph corresponding to alphas, PCNs, and "AND" nodes on a path from PCN i through branch j excluding PCN i$\}$.

Step 4: For each PCN i, construct the sets $T_i = \{(X,Y) \mid X$ is of the form S_{ijk} and Y is of the form $S_{ij'k'}$, where $j \neq j'\}$.

Step 5: Define the operator $\cap'$ such that if $A = \{a_1, a_2, \ldots, a_n\}$ and $B = \{b_1, b_2, \ldots, b_m\}$, then $A \cap' B$ is the "AND" node in A with the smallest subscript that is also an element in B.
For each PCN i and each element (C,Y) in T_i, define $R_{ixy} = X \cap' Y$. R_{ixy} is the "AND" node which synchronizes the paths X and Y.

Step 6: For each PCN i and each element (X,Y) in T_i, define $X' = \{$nodes in X corresponding to alphas between PCN i and the "AND" node $R_{ixy}\}$. Also define $Y' = \{$nodes in Y corresponding to alphas between PCN i and the "AND" node $R_{ixy}\}$. X' and Y' contain sets of alphas executable in parallel.

Step 7: For each alpha $\hat{n}$ in X', add to $h_{\hat{n}}$ all alphas in Y'. Similarly, for each alpha $\hat{n}$ in Y', add to $h_{\hat{n}}$ all alphas in X'.

Step 8: For each alpha $\hat{w}$ corresponding to a subnet, construct the set $j_{\hat{w}}$ = {Alphas in subnet $\hat{w}$ which correspond to subnets}.

Step 9: For each alpha $\hat{w}$ corresponding to a subnet, if $\hat{x} \in j_{\hat{w}}$, then set $j_{\hat{w}} = j_{\hat{w}} \cup j_{\hat{x}}$. Repeat for all $\hat{x} \in j_{\hat{w}}$ until no new elements are added to $j_{\hat{w}}$.

Step 10: For each alpha $\hat{n}$ corresponding to a subnet, let the set $h_{\hat{n}} = \bigcup_{\hat{x}_i} j_{\hat{x}_i} \cup h_{\hat{n}}$, where $\hat{x}_i \in h_{\hat{n}}$.

Step 11: For each alpha $\hat{n}$ corresponding to a subnet, where $\hat{n}$ is an element of some subnet, then let the set $h_{\hat{n}} = \bigcup_{\hat{w}} h_{\hat{w}} \cup h_{\hat{n}}$, where $\hat{n} \in j_{\hat{w}}$.

Step 12: For each module n in the program,identify M_n = {alphas in the main R-Net and subnets corresponding to module n}.

Step 13: For each module n, generate the set $H_n = \bigcup_{\hat{x} \in M_n} f(h_{\hat{x}})$, where $f(h_{\hat{x}})$ computes {modules corresponding to the alphas in $h_{\hat{x}}$}. A PIR is defined to exist between module n and each of the modules in H_n.

3.1.2 An Example for Illustrating Algorithm 3.1.1

Let us illustrate Algorithm 3.1.1 by considering the program described by the refined R-Net shown in Figure 9.

Step 1: $h_{\hat{B}} = h_{\hat{C}} = h_{\hat{D}} = h_{\hat{E}_1} = h_{\hat{H}} = h_{\hat{I}} = h_{\hat{E}_2} = h_{\hat{J}} = \emptyset$

Step 2: For the Main R-Net,

P = {PCN1,PCN2}.

Step 3: S_{111} = {B̂,D̂,AND1,Ê1,F̂,Ê2}

S_{112} = {B̂,D̂,AND1,Ê1,PCN2,(Ĥ,Î),AND2,Ê2}

S_{121} = {Ĉ,AND1,Ê1,F̂,Ê2}

S_{122} = {Ĉ,AND1,Ê1,PCN2,(Ĥ,Î),AND2,Ê2}

S_{211} = {Ĥ,AND2,Ê2}

S_{221} = {Î,AND2,Ê2} .

Step 4: Let $X_1 = X_{111}$, $X_2 = S_{112}$, $X_3 = S_{211}$

$Y_1 = S_{121}$, $Y_2 = S_{122}$, $Y_3 = S_{221}$.

Then $T_1 = \{(X_1,Y_1),(X_1,Y_2),(X_2,Y_1),(X_2,Y_2)\}$

$T_2 = \{(X_3,Y_3)\}$.

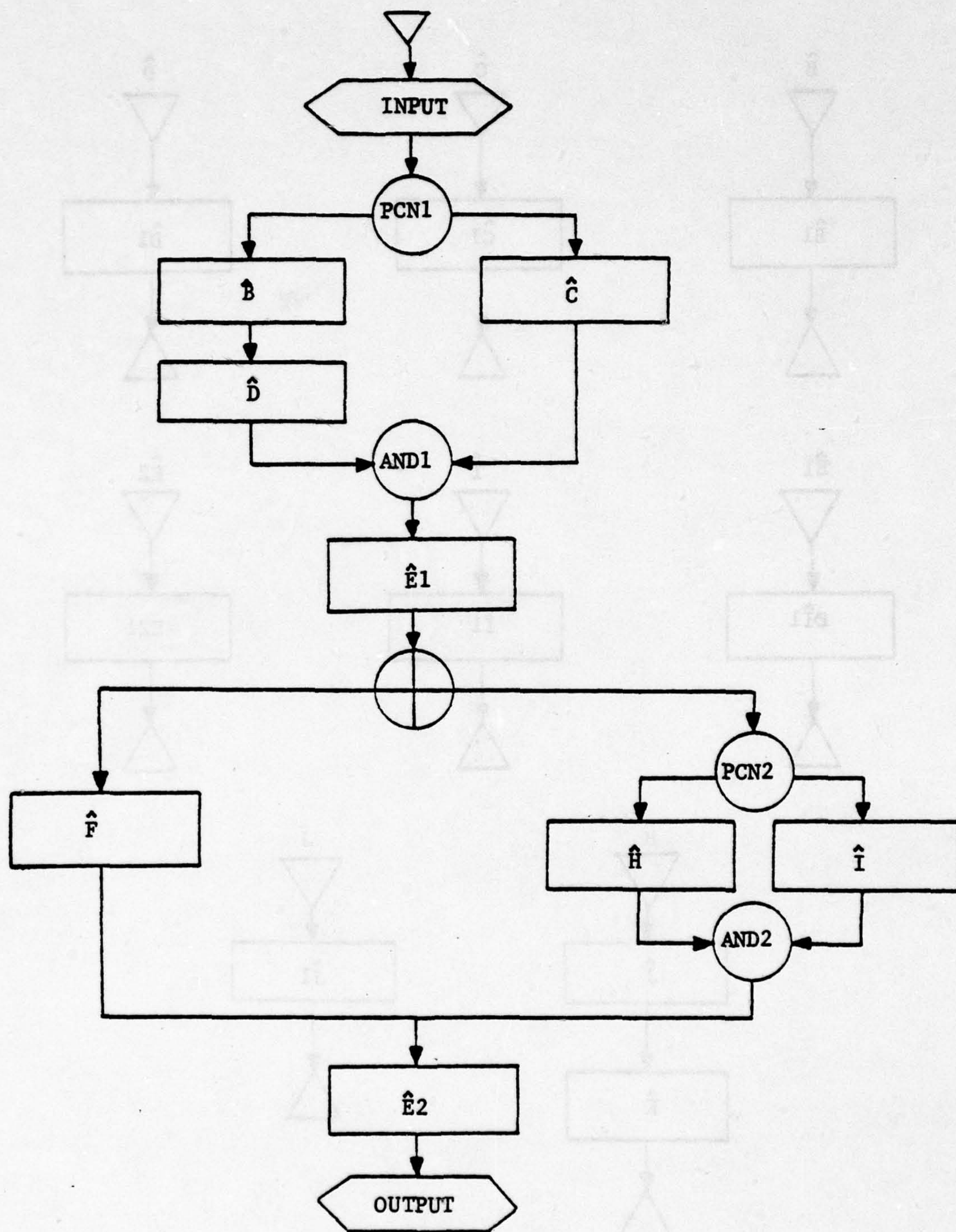


Figure 9. The refined R-Net of a program for illustrating Algorithm 3.1.1.

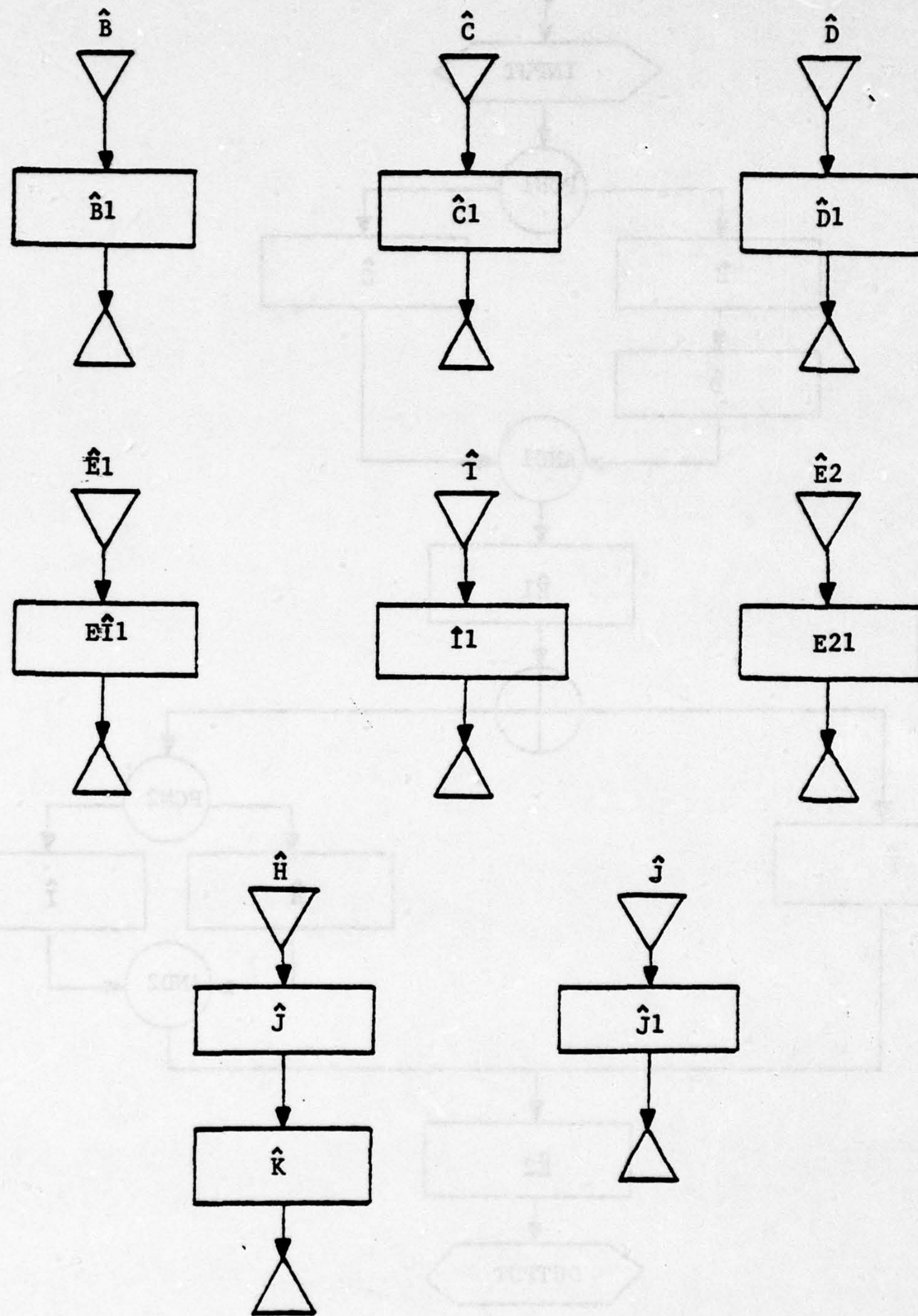


Figure 9. The refined R-Net of a program for illustrating Algorithm 3.1.1. (cont.)

Step 5: $R_1X_1Y_1$ = AND1, $R_1X_1Y_2$ = AND1,

$R_1X_2Y_1$ = AND1, $R_1X_2Y_2$ = AND1,

$R_2X_3Y_3$ = AND2.

Step 6: $X_1' = \{\hat{B},\hat{D}\}$ $X_2' = \{\hat{B},\hat{D}\}$

$X_3' = \{\hat{H}\}$.

$Y_1' = \{\hat{C}\}$, $Y_2' = \{\hat{C}\}$,

$Y_3' = \{\hat{I}\}$.

Step 7: $h_{\hat{B}} = h_{\hat{B}} \cup \{\hat{C}\}$, $h_{\hat{D}} = h_{\hat{D}} \cup \{\hat{C}\}$,

$h_{\hat{C}} = h_{\hat{C}} \cup \{\hat{B},\hat{D}\}$, $h_{\hat{H}} = h_{\hat{H}} \cup \{\hat{I}\}$,

$h_{\hat{I}} = h_{\hat{I}} \cup \{\hat{H}\}$.

None of the other subnets contain alphas executable in parallel.

Step 8: $j_{\hat{B}} = \emptyset$, $j_{\hat{C}} = \emptyset$,

$j_{\hat{D}} = \emptyset$, $j_{\hat{E}_1} = \emptyset$,

$j_{\hat{H}} = \{\hat{J}\}$, $j_{\hat{I}} = \emptyset$,

$j_{\hat{E}_2} = \emptyset$, $j_{\hat{J}} = \emptyset$.

Step 9: $j_{\hat{H}} = \{\hat{J}\} \cup \emptyset$.

Step 10: $h_{\hat{B}} = \{\hat{C}\}$, $h_{\hat{C}} = \{\hat{B},\hat{D}\}$,

$h_{\hat{D}} = \{\hat{C}\}$, $h_{\hat{E}_1} = \emptyset$,

$h_{\hat{H}} = \{\hat{I}\}$, $h_{\hat{I}} = \{\hat{J}\} \cup \{\hat{H}\} = \{\hat{H},\hat{J}\}$,

$h_{\hat{E}_2} = \emptyset$, $h_{\hat{J}} = \emptyset$.

Step 11: $h_{\hat{B}} = \{\hat{C}\}$, $h_{\hat{C}} = \{\hat{B},\hat{D}\}$,

$h_{\hat{D}} = \{\hat{C}\}$, $h_{\hat{E}_1} = \emptyset$,

$h_{\hat{H}} = \{\hat{I}\}$, $h_{\hat{I}} = \{\hat{H},\hat{J}\}$,

$h_{\hat{E}_2} = \emptyset$, $h_{\hat{J}} = \{\hat{I}\}$.

Step 12: $M_{\hat{B}} = \{\hat{B}\}$, $M_{\hat{C}} = \{\hat{C}\}$,

$M_{\hat{D}} = \{\hat{D}\}$, $M_{\hat{E}} = \{\hat{E}1, \hat{E}2\}$,

$M_{\hat{H}} = \{\hat{H}\}$, $M_{\hat{I}} = \{\hat{I}\}$,

$M_{\hat{J}} = \{\hat{J}\}$.

Step 13: $H_B = \{C\}$, $H_C = \{B, D\}$,

$H_D = \{C\}$, $H_E = \emptyset$,

$H_H = \{I\}$, $H_I = \{H, J\}$,

$H_J = \{I\}$.

A PIR is then defined to exist between the following pairs of modules: (B,C), (C,D), (H,I) and (I,J).

3.1.3 Theorem for Algorithm 3.1.1

Theorem 3.1.3: Algorithm 3.1.1 identifies all sets of modules that are executable in parallel with a given module based on the R-Nets of the program.

Proof: Let Z be an arbitrary module executable in parallel with modules $N_1, N_2, \ldots, N_k$. Step 13 of Algorithm computes the set H_Z of modules that are executable in parallel with module Z. Thus, it must be shown that if $H_z = \{Y_1, Y_2, \ldots, Y_j\}$, then J = K and $\{N_1, N_2, \ldots, N_K\} = \{Y_1, Y_2, \ldots, Y_j\}$. The proof requires 2 parts. In the first part, it must be shown that $\{Y_1, Y_2, \ldots, Y_J\} \subset \{N_1, N_2, \ldots, N_K\}$. The second part of the proof must show that $\{N_1, N_2, \ldots, N_K\} \subset \{Y_1, Y_2, \ldots, Y_j\}$.

Part 1: Establish $\{Y_1, Y_2, \ldots, Y_j\} \subset \{N_1, N_2, \ldots, N_K\}$. It must be shown that every element of $\{Y_1, Y_2, \ldots, Y_J\}$ is also an element of $\{N_1, N_2, \ldots, N_K\}$. Let Y_r be any element of $\{Y_1, Y_2, \ldots, Y_J\}$. We must consider two cases. In case 1, Y_r and Z correspond to alphas on the same R-Net. In case 2, Y_r and Z correspond to alphas on different R-Nets.

Case 1: Assume Y_r and Z correspond to alphas on the same R-Net. Now $Y_r \in H_z$ implies that there exists an alpha $\hat{Y}_r$ in the graph corresponding to module Y_r, a PCN i, and an element (X,Y) in T_i such that $\hat{Y}_r \in Y'$ and $\hat{Z} \in X'$ by Steps 6, 7, 12, and 13 of the algorithm, where $\hat{Z}$ is an alpha in the graph

corresponding to module Z. This implies $\hat{Y}_r$ and $\hat{Z}$ are on separate paths from PCN i and are located on the paths before R_{ixy} by Steps 3, 4 5, and 6. Since R_{ixy} is the "AND" node for synchronizing the paths, by definition of the R-Net, $\hat{Y}_r$ and $\hat{Z}$ are executable in parallel. Thus, modules Y_r and Z are executable in parallel by Steps 12 and 13.

Case 2: Assume Y_r and Z correspond to alphas on different R-Nets. Now $Y_r \in H_z$ implies that there exists alphas $\hat{Y}_r$ corresponding to module Y_r and $\hat{u}$ such that $\hat{Y}_r \in j_u$ and $\hat{u} \in h_{\hat{z}}$ where $\hat{Z}$ is an alpha corresponding to module Z and $\hat{u}$ is on the same R-Net as $\hat{Z}$ by Steps 10, 12, and 13 of the algorithm. This implies there exists a module u corresponding to $\hat{u}$ on the same R-Net as Z with $u \in H_Z$ by Steps 12 and 13 of the algorithm. By Case 1 of the proof, we then have u is executable in parallel with module Z. Now $\hat{Y}_r \in j_{\hat{u}}$ implies by Steps 8, 9, 12, and 13 that module Y_r is invoked directly or indirectly by module u. But since module u is executable in parallel with module Z and Y_r is invoked directly or indirectly by module u, module Y_r must be executable in parallel with module Z.

Thus, in either Case 1 or Case 2, Y_r is executable in parallel with module Z and this implies $Y_r \in \{N_1, N_2, \ldots, N_k\}$. Hence, $\{Y_1, y_2, \ldots, Y_j\} \subset \{N_1, N_2, \ldots, N_k\}$.

Part 2: Establish $\{N_1, N_2, \ldots, N_K\} \subset \{Y_1, Y_2, \ldots, Y_J\}$. It must be shown that every element of $\{N_1, N_2, \ldots, N_K\}$ is also an element of $\{Y_1, Y_2, \ldots, Y_J\}$. Let N_r be any element of $\{N_1, N_2, \ldots, N_K\}$. Now $N_r \in \{N_1, N_2, \ldots, N_K\}$, which implies that N_r is executable in parallel with module Z. We must consider two cases. In Case 1, Y_r and Z correspond to alphas on the same R-Net. In case 2, Y_r and Z correspond to alphas on different R-Nets.

Case 1: Assume N_r and Z correspond to alphas on the same R-Net. Then by definition of the R-Net structure representing the program, this implies that there exists a PCN i with one branch that contains an alpha $\hat{Z}$ corresponding to module Z and another branch that contains an alpha $\hat{N}_r$ corresponding to module N_r with both $\hat{Z}$ and $\hat{N}_r$ preceding the "AND" node which synchronizes the paths. This implies that there exists an element (X,Y) in T_i such that $\hat{N}_r \in X'$ and $\hat{Z} \in Y'$ by Steps 4, 5, and 6 of the algorithm. Step 7 then guarantees that $\hat{N}_r \in h_{\hat{z}}$. Steps 12 and 13 then guarantee $N_r \in H_z = \{Y_1, Y_2, \ldots, Y_J\}$.

Case 2: Assume N_r and Z correspond to alphas on different R-Nets. Now N_r is executable in parallel with module Z implies that there exists a module u which invokes directly or indirectly N_r such that u is executable in parallel with Z and u and Z correspond to alphas on the same R-Net. By Case 1 of Part 2 of the proof, this implies $u \in H_Z$. Steps 12 and 13 of the algorithm then imply $\hat{u} \in H_{\hat{Z}}$, where $\hat{u}$ and $\hat{Z}$ are alphas corresponding to modules u and Z respectively. Now since u invokes directly or indirectly N_r, there exists an alpha $\hat{N}_r$ corresponding to N_r and such that $N_r \in j_{\hat{u}}$ by Steps 8, 9, 12, and 13. Then, since $\hat{u} \in H_{\hat{Z}}$ and $\hat{N}_r \in j_{\hat{u}}$, $\hat{N}_r \in H_{\hat{Z}}$ by Step 10. Steps 12 and 13 then guarantee $N_r \in H_Z = \{Y_1, Y_2, \ldots, Y_J\}$.

Thus, in either Case 1 or Case 2, $N_r \in H_Z = \{Y_1, Y_2, \ldots, Y_J\}$, and $\{N_1, N_2, \ldots, N_K\} \subset \{Y_1, Y_2, \ldots, Y_J\}$. Since $\{N_1, N_2, \ldots, N_K\} \subset \{Y_1, Y_2, \ldots, Y_J\}$ and $\{Y_1, Y_2, \ldots, Y_J\} \subset \{N_1, N_2, \ldots, N_K\}$, J must equal K and $\{N_1, N_2, \ldots, N_K\} = \{Y_1, Y_2, \ldots, Y_J\}$.

3.1.4 Identification of Parallel Execution Sets

In addition to the identification of the pairs of modules executable in parallel accomplished by Algorithm 3.1.1, in large and complex programs, it is also important to determine sets consisting of all modules that can be executed in parallel. These sets can be defined as <u>parallel execution sets</u>. An example of a larger and more complex system is illustrated in Figure 6. One example of a parallel execution set from this example is {E,F,C,I,J} since it is possible for each of the modules in the set to be executed in parallel. Parallel execution sets play an important role in the resource contention problem, since it is necessary to determine how many modules are competing for the fixed number of available resources. Although the modules in parallel execution sets may seldomly be executed simultaneously in the system due to their current relative execution starting time, the potential for their simultaneous execution exists. After a period of operation and maintenance, it is possible that the modules in the parallel execution sets could be executed simultaneously and, therefore, this consideration should remain a factor in determining the effect of software modifications. Since the identification of the parallel execution sets is an important step in many of the algorithms for identifying the mechanisms for the propagation of performance changes, a common algorithm has been developed for accomplishing

this objective.

An algorithm to identify the parallel execution sets in the program will now be briefly discussed. The first step of the algorithm identifies the pairs of modules executable in parallel. This is accomplished by invoking Algorithm 3.1.1. After each module in the system has had its set of modules that can be executed in parallel with it identified, the parallel execution sets can then be formed. These sets are easy to create. The first step is the selection of a module and its set of modules that can be executed in parallel with it. At this point, there are at least two modules that can be executed simultaneously. Next, a module is selected within the set of parallel executable modules. Its corresponding set of parallel executable modules is then intersected with the remaining modules in the set to determine if there exists three modules that can be executed simultaneously. The process is iterated for all modules remaining in the set and for all modules in the system.

There are several important parameters pertaining to software maintainability from a performance perspective that can be gleaned from the output of this algorithm. For example, the <u>degree</u> of <u>parallel influence</u> of a module can be defined as the number of modules which can be executed in parallel with it. The degree of parallel influence can serve as a measure of the complexity of modification of a module since it measures the number of potential performance changes in modules that can occur as a result of a modification.

An algorithm for identifying the parallel execution sets in the program will now be formally described.

3.1.5 Algorithm to Identify the Parallel Execution Sets in a Program

Algorithm 3.1.5

Step 1: For each module n, construct sets $h_{\hat{n}_i}$ where $\hat{n}_i$ are alphas in the graph corresponding to module n via Algorithm 3.1.1.

Step 2: Select an alpha $\hat{n}_i$. If all alphas have been selected, go to Step 9.

Step 3: If $h_{\hat{n}_i} = \emptyset$, then set $h'_{\hat{n}_i} = \emptyset$ and go to Step 2, else set $h'_{\hat{n}_i} = \{\{y_i\}\}$ where $y_i \in h_{\hat{n}_i}$. $h'_{\hat{n}_i}$ will contain the set of parallel execution sets for alpha $\hat{n}_i$.

Step 4: Initialize $j = 1$.

Step 5: Select an element, $z = \{\hat{x}_1, \hat{x}_2, \ldots, \hat{x}_j\}$, of $h'_{\hat{n}_i}$ of size j. If all elements of size j have been selected, go to Step 8.

Step 6: Compute $h_{\hat{x}_1} \cap h_{\hat{x}_2} \ldots \cap h_{\hat{x}_j} \cap h_{\hat{n}_i}$. If the intersection is empty, go to Step 5.

Step 7: For each $\hat{w} \in (h_{\hat{x}_1} \cap h_{\hat{x}_2} \ldots \cap h_{\hat{x}_j} \cap h_{\hat{n}_i})$ compute $h'_{\hat{n}_i} = h'_{\hat{n}_i} \cup \{z \cup \hat{w}\}$. Go to Step 5.

Step 8: $j = j + 1$. If there are no new elements in $h'_{\hat{n}_i}$ of size j, go to Step 2, else go to Step 5.

Step 9: For each module n in the program, identify $M_n = \{$alphas in the graph corresponding to module n$\}$.

Step 10: For each of these modules n, set $H'_n = \bigcup_{\hat{x} \in M_n} f(h'_{\hat{x}})$, where $f(h'_{\hat{x}})$ computes the sets of modules in $h'_{\hat{x}}$ corresponding to the sets of alphas in $h'_{\hat{x}}$.

3.1.6 An Example for Illustrating Algorithm 3.1.5

Let us use the program shown in Figure 10 to illustrate Algorithm 3.1.5.

Step 1: $h_{\hat{A}1} = \emptyset$ $\qquad h_{\hat{B}} = \{\hat{C}, \hat{F}\}$

$h_{\hat{C}} = \{\hat{B}, \hat{D}, \hat{E}\}$ $\qquad h_{\hat{D}} = \{\hat{E}, \hat{C}, \hat{F}\}$

$h_{\hat{E}} = \{\hat{D}, \hat{C}, \hat{F}\}$ $\qquad h_{\hat{F}} = \{\hat{B}, \hat{D}, \hat{E}\}$

$h_{\hat{A}2} = \emptyset$.

Step 2: Consider alpha $\hat{A}1$.

Step 3: Since $h_{\hat{A}1} = \emptyset$, $h'_{\hat{A}1} = \emptyset$.

Step 2: Consider alpha $\hat{B}$.

Step 3: $h'_{\hat{B}} = \{\{\hat{C}\}, \{\hat{F}\}\}$.

Step 4: $j = 1$.

Step 5: $z = \{\hat{C}\}$.

Step 6: $h_{\hat{C}} \cap h_{\hat{B}} = \emptyset$.

Step 5: $z = \{\hat{F}\}$.

Step 6: $h_{\hat{F}} \cap h_{\hat{B}} = \emptyset$.

Step 8: $j = 2$.

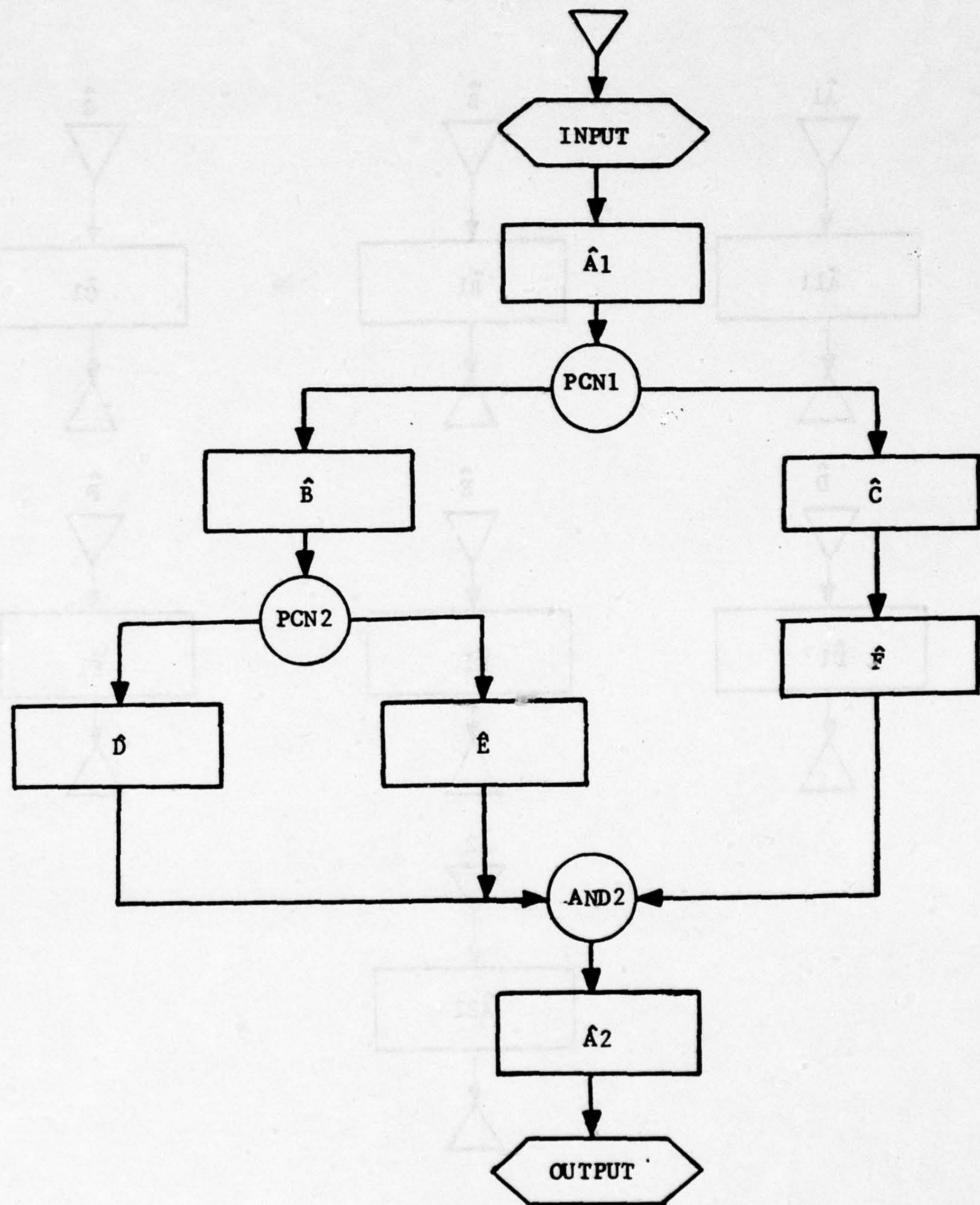


Figure 10. An example for illustrating Algorithm 3.1.5.

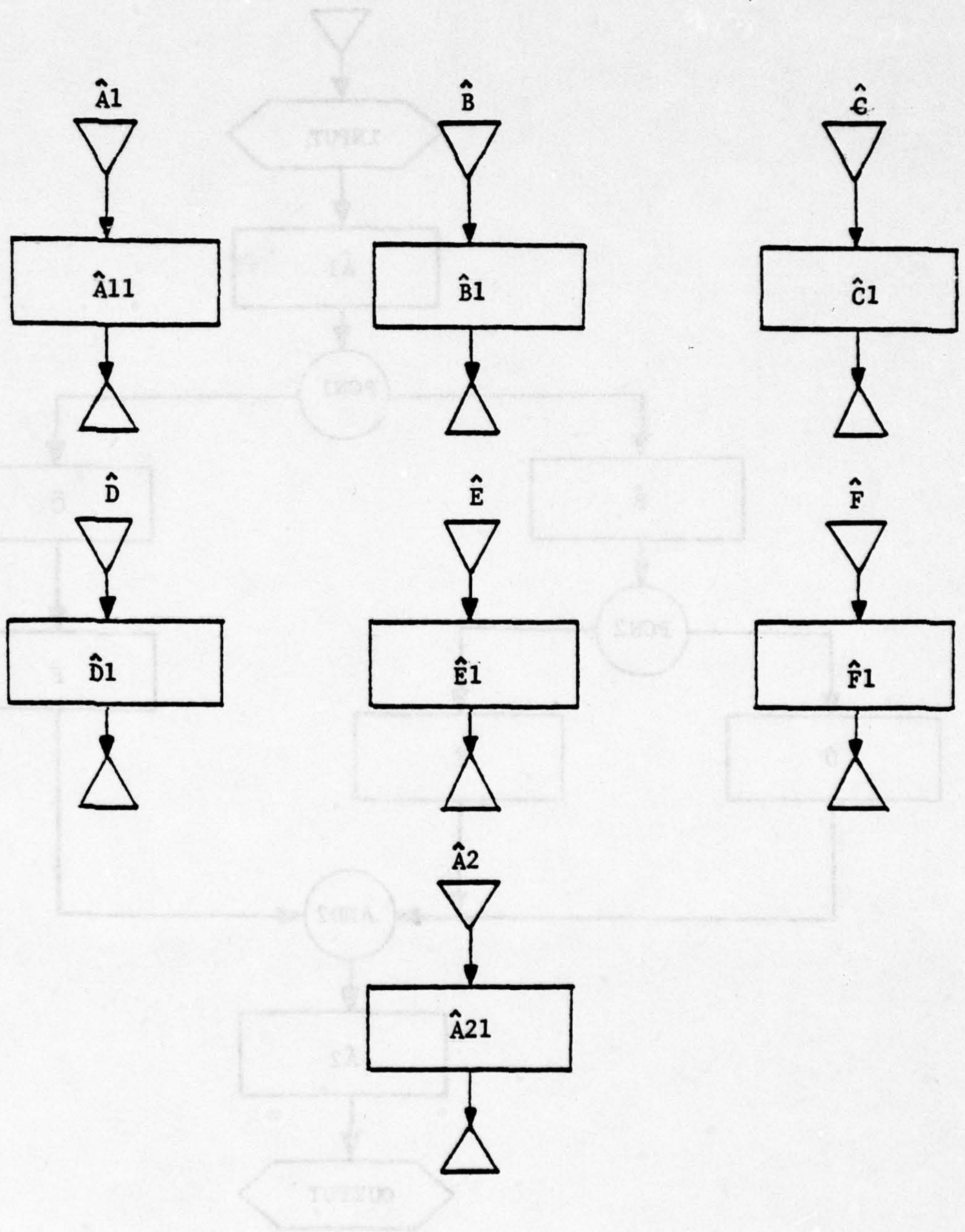


Figure 10. An example for illustrating Algorithm 3.1.5 . (cont.)

Step 2: Consider alpha $\hat{C}$.

Step 3: $h'_{\hat{C}} = \{\{\hat{B}\}, \{\hat{D}\}, \{\hat{E}\}\}$.

Step 4: $j = 1$.

Step 5: $z = \{\hat{B}\}$.

Step 6: $h_{\hat{B}} \cap h_{\hat{C}} = \emptyset$.

Step 5: $z = \{\hat{D}\}$.

Step 6: $h_{\hat{D}} \cap h_{\hat{C}} = \{\hat{E}\}$.

Step 7: $h'_{\hat{C}} = h'_{\hat{C}} \cup \{\{\hat{E},\hat{D}\}\}$

Step 5: $z = \{\hat{E}\}$.

Step 6: $h_{\hat{E}} \cap h_{\hat{C}} = \{\hat{D}\}$.

Step 7: $h'_{\hat{C}} = h'_{\hat{C}} \cup \{\{\hat{E},\hat{D}\}\}$.

Step 8: $j = 2$.

Step 5: $z = \{\hat{D},\hat{E}\}$.

Step 6: $h_{\hat{D}} \cap h_{\hat{E}} \cap h_{\hat{C}} = \emptyset$.

Step 8: $j = 3$.

Step 2: Consider alpha $\hat{D}$.

Step 3: $h'_{\hat{D}} = \{\{\hat{E}\}, \{\hat{C}\}, \{\hat{F}\}\}$.

Step 4: $j = 1$.

Step 5: $z = \{\hat{E}\}$

Step 6: $h_{\hat{E}} \cap h_{\hat{D}} = \{\hat{C},\hat{F}\}$

Step 7: $h'_{\hat{D}} = h'_{\hat{D}} \cup \{\{\hat{E},\hat{C}\}, \{\hat{E},\hat{F}\}\}$

Step 5: $z = \{\hat{C}\}$

Step 6: $h_{\hat{C}} \cap h_{\hat{D}} = \{\hat{E}\}$

Step 7: $h'_{\hat{D}} = h'_{\hat{D}} \cup \{\{\hat{C},\hat{E}\}\}$

Step 5: $z = \{\hat{F}\}$

Step 6: $h_{\hat{F}} \cap h_{\hat{D}} = \{\hat{E}\}$

Step 7: $h'_{\hat{D}} = h'_{\hat{D}} \cup \{\{\hat{F},\hat{E}\}\}$

Step 8: $j = 2$

Step 5: $z = \{\hat{E},\hat{C}\}$

Step 6: $h_{\hat{E}} \cap h_{\hat{C}} \cap h_{\hat{D}} = \emptyset$

Step 5: $z = \{\hat{E},\hat{F}\}$

Step 6: $h_{\hat{E}} \cap h_{\hat{F}} \cap h_{\hat{D}} = \emptyset$

Step 8: $j = 3$

Step 2: Consider alpha $\hat{E}$

Step 3: $h'_{\hat{E}} = \{\{\hat{D}\}, \{\hat{C}\}, \{\hat{F}\}\}$

Step 4: $j = 1$

Step 5: $z = \{\hat{D}\}$

Step 6: $h_{\hat{D}} \cap h_{\hat{E}} = \{\hat{C}, \hat{F}\}$

Step 7: $h'_{\hat{E}} = h'_{\hat{E}} \cup \{\{\hat{D}, \hat{C}\}, \{\hat{D}, \hat{F}\}\}$

Step 5: $z = \{\hat{C}\}$

Step 6: $h_{\hat{C}} \cap h_{\hat{E}} = \{\hat{D}\}$

Step 7: $h'_{\hat{E}} = h'_{\hat{E}} \cup \{\{\hat{C}, \hat{D}\}\}$

Step 5: $z = \{\hat{F}\}$

Step 6: $h_{\hat{F}} \cap h_{\hat{E}} = \{\hat{D}\}$

Step 7: $h'_{\hat{E}} = h'_{\hat{E}} \cup \{\{\hat{F}, \hat{D}\}\}$

Step 8: $j = 2$

Step 5: $z = \{\hat{C}, \hat{D}\}$

Step 6: $h_{\hat{C}} \cap h_{\hat{D}} \cap h_{\hat{E}} = \emptyset$

Step 5: $Z = \{\hat{D}, \hat{F}\}$

Step 6: $h_{\hat{D}} \cap h_{\hat{F}} \cap h_{\hat{E}} = \emptyset$

Step 8: $j = 3$

Step 2: Consider alpha $\hat{F}$

Step 3: $h'_{\hat{F}} = \{\{\hat{B}\}, \{\hat{D}\}, \{\hat{E}\}\}$

Step 4: $j = 1$

Step 5: $z = \{\hat{B}\}$

Step 6: $h_{\hat{B}} \cap h_{\hat{F}} = \emptyset$

Step 5: $z = \{\hat{D}\}$

Step 6: $h_{\hat{D}} \cap h_{\hat{F}} = \{\hat{E}\}$

Step 7: $h'_{\hat{F}} = h'_{\hat{F}} \cup \{\{\hat{D}, \hat{E}\}\}$

Step 5: $z = \{\hat{E}\}$

Step 6: $h_{\hat{E}} \cap h_{\hat{F}} = \{\hat{D}\}$

Step 7: $h'_{\hat{F}} = h'_{\hat{F}} \cup \{\{\hat{E}, \hat{D}\}\}$

Step 8: $j = 2$

Step 5: $z = \{\hat{D}, \hat{E}\}$

Step 6: $h_{\hat{D}} \cap h_{\hat{E}} \cap h_{\hat{F}} = \emptyset$

Step 8: $j = 3$

Step 2: Consider alpha $\hat{A}2$

Step 3: Since $h_{\hat{A}2} = \emptyset$, $h'_{\hat{A}2} = \emptyset$

When the algorithm reaches Step 9, we have:

$$h'_{\hat{A}1} = \emptyset$$

$$h'_{\hat{B}} = \{\{\hat{C}\},\{\hat{F}\}\}$$

$$h'_{\hat{C}} = \{\{\hat{B}\},\{\hat{D}\}\ \{\hat{E}\},\{\hat{D},\hat{E}\}\}$$

$$h'_{\hat{D}} = \{\{\hat{E}\},\{\hat{C}\},\{\hat{F}\},\{\hat{E},\hat{C}\},\{\hat{E},\hat{F}\}\}$$

$$h'_{\hat{E}} = \{\{\hat{D}\},\{\hat{C}\},\{\hat{F}\},\{\hat{C},\hat{D}\},\{\hat{D},\hat{F}\}\}$$

$$h'_{\hat{F}} = \{\{\hat{B}\},\{\hat{D}\},\{\hat{E}\},\{\hat{D},\hat{E}\}\}$$

$$h'_{\hat{A}2} = \emptyset$$

Step 9: $M_A = \{\hat{A}1,\hat{A}2\}$, $M_B = \{\hat{B}\}$,

$M_C = \{\hat{C}\}$, $M_D = \{\hat{D}\}$,

$M_E = \{\hat{E}\}$, $M_F = \{\hat{F}\}$

Step 10: $H'_A = \emptyset$,

$$H'_B = \{\{C\},\{F\}\}$$

$$H'_C = \{\{B\},\{D\},\{E\},\{D,E\}\}$$

$$H'_D = \{\{E\},\{C\},\{F\},\{E,C\},\{E,F\}\}$$

$$H'_E = \{\{D\},\{C\},\{F\},\{C,D\},\{D,F\}\}$$

$$H'_F = \{\{B\},\{D\},\{E\},\{D,E\}\}$$

3.1.7 Theorem for Algorithm 3.1.5

Theorem 3.1.7: Algorithm 3.1.5 computes the parallel execution sets for a program.

Proof: Let n be any module in the program. The following must be shown:

CRITERION 1: All elements of H'_n are sets of modules executable in parallel at the same time as module n.

CRITERION 2: Every set of modules executable in parallel at the same time as module n is an element of H'_n.

Both Criterion 1 and Criterion 2 will be shown to be satisfied by induction.

<u>Proof of Criterion 1 by Induction</u>: Consider an arbitrary element, $\{y\}$, of H'_n of size 1. Steps 3, 8, and 9 of Algorithm 3.1.5 guarantee that there exists an alpha $\hat{n}_i$ corresponding to module n and an alpha $\hat{y}$ corresponding to module y such that $\hat{y} \in h_{\hat{n}_i}$. By definition of $h_{\hat{n}_i}$, $\hat{y}$ is executable in parallel with alpha $\hat{n}_i$. Steps 9 and 10 then imply module y is executable in parallel with module n. Thus, CRITERION 1 is true for all elements of H'_n of size 1.

Now assume all elements of size j in H'_n satisfy Criterion 1. It must be shown that all elements of size j + 1 also satisfy Criterion 1.

Let $z = \{x_1, x_2, \ldots, x_j\}$ be an arbitrary element of H'_n of size j. This implies that there exists alphas $\hat{x}_1, \hat{x}_2, \ldots, \hat{x}_j$ corresponding to modules $x_1, x_2, \ldots, x_j$. Steps 6 and 7 of the algorithm select an alpha $\hat{w}$ such that $\hat{w} \in h_{\hat{x}_1} \cap h_{\hat{x}_2} \ldots \cap h_{\hat{x}_j} \cap h_{\hat{n}_i}$. The element $y = \{\hat{x}_1, \hat{x}_2, \ldots, \hat{x}_j, \hat{w}\}$ of size j + 1 is then added to $h_{\hat{n}_i}'$. Now since $\hat{w} \in (h_{\hat{x}_1} \cap h_{\hat{x}_2} \ldots \cap h_{\hat{x}_j} \cap h_{\hat{n}_i})$, it implies $\hat{w}$ is executable in parallel at the same time as alphas $\hat{x}_1, \hat{x}_2, \ldots, \hat{x}_j, \hat{n}_i$. Steps 9 and 10 then imply that there exists a module w corresponding to $\hat{w}$ which is executable in parallel with modules $x_1, x_2, \ldots, x_j, n$. Thus, y satisfies Criterion 1. Therefore, all elements of size j + 1 satisfy Criterion 1.

By induction, this implies all elements of H'_n of all sizes satisfy CRITERION 1

<u>Proof of Criterion 2 by Induction</u>: It must be shown that all sets of modules executable in parallel at the same time as module n of a given size are elements of H'_n.

Consider sets of elements of size 1. Let y be an arbitrary module executable in parallel with module n. This implies that there exists an alpha $\hat{y}$ corresponding to module y and an alpha $\hat{n}_i$ corresponding to module n such that $\hat{y} \in h_{\hat{n}_i}$. Now Step 3 of the algorithm adds $\{\hat{y}\}$ to $h'_{\hat{n}_i}$. Steps 9 and 10 then insure that $\{y\}$ is added to H'_n. Thus, all sets of modules executable in parallel at the same time as module n of size 1 are elements of H'_n.

Now assume all sets of modules executable in parallel with module n of size j are elements of H'_n. Let $z = \{x_1, x_2, \ldots, x_j, x_{j+1}\}$ be a set of modules executable in parallel with module n. This implies that there exist alphas $\hat{x}_1, \hat{x}_2, \ldots, \hat{x}_j, \hat{x}_{j+1}$ corresponding to modules $x_1, x_2, \ldots, x_j, x_{j+1}$. Let $\hat{z} = \{\hat{x}_1, \hat{x}_2, \ldots, \hat{x}_j, \hat{x}_{j+1}\}$. Now we have assumed that there exists some subset $w = \{x_1, x_2, \ldots, x_j\}$ of size j of z which is an element of H'_n. This implies that there exists a subset $\hat{w} = \{\hat{x}_1, \hat{x}_2, \ldots, \hat{x}_j\}$ of $\hat{z}$ of size j which is an element of $h'_{\hat{n}_i}$. $\hat{x}_{j+1} \in \hat{z}$ implies x_{j+1} is executable in parallel with each element of $\{\hat{x}_1, \hat{x}_2, \ldots, \hat{x}_j, \hat{x}_{n_i}\}$, where n_i is an alpha corresponding to module n. This implies $\hat{x}_{j+1} \in (h_{\hat{x}_1} \cap h_{\hat{x}_2} \ldots \cap h_{\hat{x}_j} \cap h_{\hat{x}_{n_i}})$. Step 7 of the algorithm then implies $\hat{z} \in H'_n$. Steps 9 and 10 of the algorithm then imply $z \in H'_n$. Therefore, all sets of modules executable in parallel at the same time as module n of size j + 1 are elements of H'_n.

Therefore, every set of modules executable in parallel at the same time as module n are elements of H'_n.

3.2 Shared Resources

The next mechanism for the propagation of performance changes to be formally described is the shared resources mechanism. When modules are forced to share resources, the time when each module requests and releases common resources are important performance parameters. Thus, software modifications producing performance changes in the time resources are utilized can have detrimental effects on the performance of modules that must also share the resources.

Performance attributes 2, 3, and 4 are associated with this mechanism. The shared resources mechanism can be defined in terms of the following performance dependency rules:

MODULE X/PA.2 for resource i → MODULE Y/PA.2 for resource i if module X is involved in a PIR with module Y via mechanism 2.

MODULE X/PA.2 for resource i → MODULE Y/PA.3 for resource i if module X = module Y or module X is involved in a PIR with module Y via mechanism 2.

MODULE X/PA.2 for resource i → MODULE Y/PA.4 if module X is involved in a PIR with module Y via mechanism 2.

MODULE X/PA.2 for resource i → MODULE Y/PA.7 for resource i if module X = module Y.

MODULE X/PA.3 for resource i → MODULE Y/PA.2 for resource i if module X is involved in a PIR with module Y via mechanism 2.

MODULE X/PA.3 for resource i → MODULE Y/PA.3 for resource i if module X is involved in a PIR with module Y via mechanism 2.

MODULE X/PA.3 for resource i → MODULE Y/PA.4 if module X is involved in a PIR with module Y via mechanism 2.

MODULE X/PA.3 for resource i → MODULE Y/PA.7 for resource i if module X = module Y.

MODULE X/PA.4 → MODULE Y/PA.2 for resource i if module X = module Y.

MODULE X/PA.4 → MODULE Y/PA.3 for resource i if module X = module Y.

MODULE X/PA.4 → MODULE Y/PA.4 if there exists a module Z such that a PIR is in existence between module Y and module Z via mechanism 2 and module X has precedence over module Y. Module X is defined to have precedence over module Y if module X is a successor of the PCN controlling the parallel execution of modules X and Z and a predecessor of module Y on a path from the PCN.

MODULE X/PA.6 → MODULE Y/PA.4 if there exists a module Z such that a PIR is in existence between module Y and module Z via mechanism 2 and module X has precedence over module Y.

MODULE X/PA.2 for resource i → MODULE Y/VPA.1 for the request for resource i if module X is involved in a PIR with module Y via mechanism 2.

MODULE X/PA.3 for resource i → MODULE Y/VPA.1 for the request for resource i if module X is involved in a PIR with module Y via mechanism 2.

An algorithm has been developed for identifying the existence of the shared resources mechanism in a program and the modules in a program affected

by this mechanism. To accomplish this objective, modules sharing a resource that can be executed in parallel must be identified by this algorithm. For example, if modules A and B must share a resource, a modification of resource utilization in module A will not have an effect on module B if module A must complete execution before module B can begin. Only if the execution of module A and module B overlap can the modification have an effect on performance. The modules sharing common resources and executable in parallel are then identified as being part of a performance interdependency relationship since a modification to one of the modules affecting its resource utilization can affect the performance of the other modules.

An algorithm for identifying the shared resources mechanism in a program will now be formally described.

3.2.1 Algorithm to Identify the Shared Resources Mechanism

Algorithm 3.2.1

Step 1: For each resource i, set T_i initially as T_i = {all modules utilizing resource i}.

Step 2: For each module in T_i not on the main R-Net, construct S_i = {all modules invoking modules in T_i}.

Step 3: Construct $T_i' = T_i \cup S_i$. If $T_i' \neq T_i$, then set $T_i = T_i'$ and go to Step 2; otherwise go to Step 4.

Step 4: Identify sets H_n' of modules executable in parallel at the same time as module n utilizing Algorithm 3.1.5.

Step 5: For all modules n and for each i corresponding to a resource and each j corresponding to an element in H_n', construct the set $X_{inj} = T_i \cap ((j\text{th element of } H_n') \cup \{n\})$. The modules in each set X_{inj} may be in contention for resource i.

Step 6: Compute $|X_{inj}|$, i.e. the number of modules in X_{inj} for all i, n and j. If $|X_{inj}|$ is greater than the number of resources of type i, then a PIR exists among the modules in X_{inj}. The PIR is stored along with an identification of the resource in contention.

3.2.2 An Example for Illustrating Algorithm 3.2.1

To illustrate Algorithm 3.2.1, let us consider the program whose refined R-Net is shown in Figure 11.

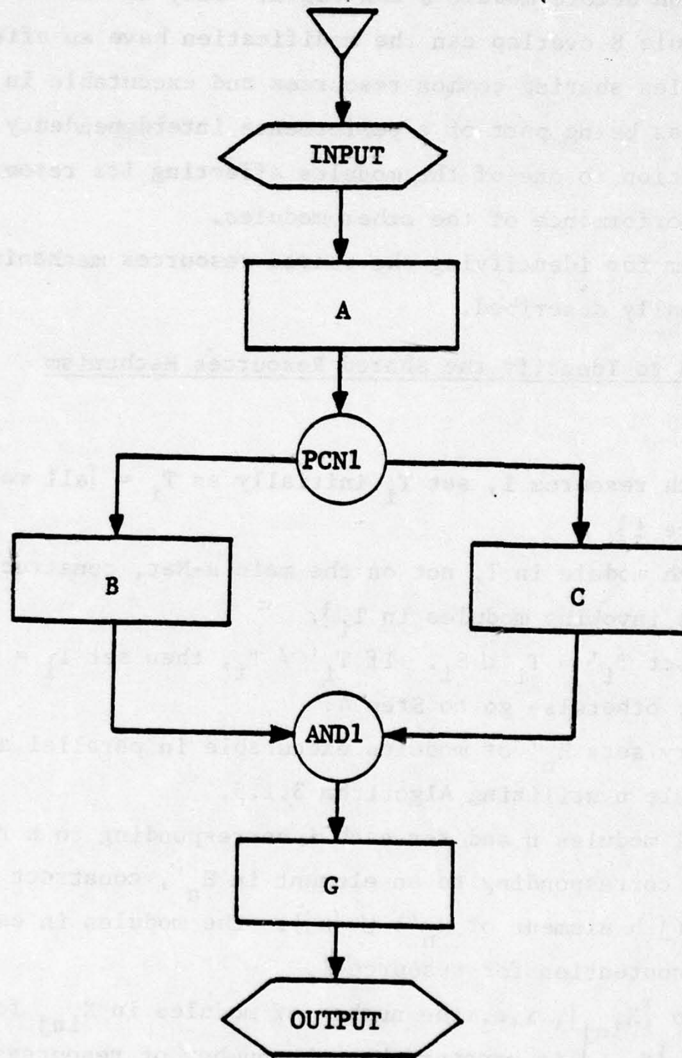


Figure 11. An example for illustrating Algorithm 3.2.1.

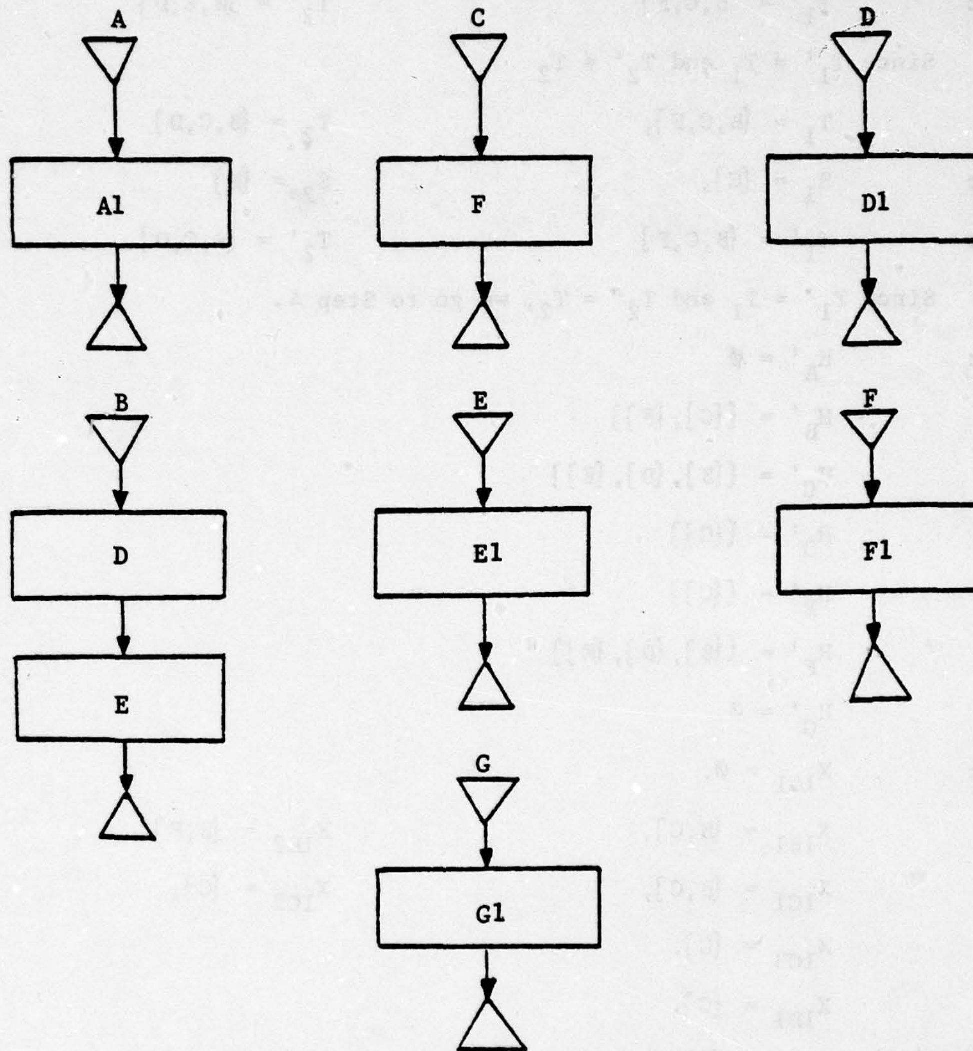


Figure 11. An example for illustrating Algorithm 3.2.1. (cont.)

Step 1: $T_1 = \{B,F\}$, $T_2 = \{D,C\}$.

Step 2: $S_1 = \{C\}$, $S_2 = \{B\}$

Step 3: $T_1' = B,C,F\}$ $T_2' = \{B,C,D\}$

Since $T_1' \neq T_1$ and $T_2' \neq T_2$

$T_1 = \{B,C,F\}$, $T_2 = \{B,C,D\}$

Step 2: $S_1 = \{C\}$, $S_2 = \{B\}$

Step 3: $T_1' = \{B,C,F\}$ $T_2' = \{B,C,D\}$

Since $T_1' = T_1$ and $T_2' = T_2$, we go to Step 4.

Step 4: $H_A' = \emptyset$

$H_B' = \{\{C\}, \{F\}\}$

$H_C' = \{\{B\}, \{D\}, \{E\}\}$

$H_D' = \{\{C\}\}$

$H_E' = \{\{C\}\}$

$H_F' = \{\{B\}, \{D\}, \{E\}\}$

$H_G' = \emptyset$

Step 5: $X_{1A1} = \emptyset$.

$X_{1B1} = \{B,C\}$, $X_{1B2} = \{B,F\}$.

$X_{1C1} = \{B,C\}$, $X_{1C2} = \{C\}$,

$X_{1C3} = \{C\}$.

$X_{1D1} = \{C\}$.

$X_{1E1} = \{C\}$.

$X_{1F1} = \{B,F\}$, $X_{1F2} = \{F\}$,

$X_{1F3} = \{F\}$.

$X_{1G1} = \emptyset$.

$X_{2A1} = \emptyset$.

$X_{2B1} = \{B,C\}$, $X_{2B2} = \{B\}$.

$X_{2C1} = \{B,C\}$ $X_{2C2} = \{C,D\}$,

$X_{2C3} = \{C\}$.

$X_{2D1} = \{C,D\}$.

$X_{2E1} = \{C\}$.

$X_{2F1} = \{B\}$, $X_{2F2} = \{D\}$,

$X_{2F3} = \emptyset$.

$X_{2G1} = \emptyset$.

Step 6: $|X_{1B1}| = |X_{1B2}| = |X_{1C1}| = |X_{1F1}| = 2$. Therefore, a PIR exists between module B and module F since resource 1 exists in quantity 1. $|X_{2B1}| = |X_{2C1}| = |X_{2D1}| = 2$. Therefore, a PIR exists between module B and module C and between module C and module D since resource 2 exists in quantity 1.

3.2.3 Theorem for Algorithm 3.2.1

Theorem 3.2.3. Algorithm 3.2.1 computes sets of all modules which may be in contention for a resource.

Proof: Let X_{inj} be an arbitrary set of modules which may be in contention for resource i. The following must be shown to be true:

(1) Any module in X_{inj} may be in contention for resource i.

(2) Any set of modules in contention for resource i is contained in some set X_{inj}.

The proof of the theorem requires proofs of both (1) and (2).

Proof of (1): Let y be an arbitrary module in X_{inj}. This implies that $y \in T_i$ and $y \in$ ((jth element of H_n') $\cup \{n\}$). Now $y \in T_i$ implies y utilizes resource i. Also, since X_{inj} is a set of modules in contention for resource i, $|X_{inj}|$ must be greater than the number of resources of type i. This implies $|X_{inj}| \geq 2$. Thus, there exists at least one other module, z, which is also an element of X_{inj}. Now $j \in X_{inj}$ and $z \in X_{inj}$ imply that y and z are

elements of the ((jth element of H_n') ∪ {n}). This in turn implies that y and z may be executed at the same time.

Therefore, y is in contention for resource i with at least module z.

Proof of (2): Let $z = \{y_1, y_2, \ldots, y_j\}$ be a set of modules in contention for resource i. This implies $y_1, y_2, \ldots, y_j$ are elements of T_i by definition of T_i. Since $y_1, y_2, \ldots, y_j$ are executable in parallel, $z - \{y_1\}$ is a subset of some element k in H'_{y_1} by definition of H'_{y_1}. This implies that z is contained in the set X_{iy_1k}.

3.3 Interprocess Communication

The next mechanism for the propagation of performance changes to be formally described is the interprocess communication mechanism. When one module must send a message to another module, the performance of the module receiving the message depends upon when the message is actually received. Thus, modifications to the module sending the message that alter the time when the message is sent can affect the performance of the module designated to receive the message.

Performance attributes 4 and 5 are associated with this mechanism. The interprocess communication mechanism can be defined in terms of the following performance dependency rules:

> MODULE X/PA.4 → MODULE Y/PA.4 if there exists a module Z such that a PDR is in existence between module Y and module Z via mechanism 3 and module X has precedence over module Y.
>
> MODULE X/PA.4 → MODULE Y/PA.5 for message i if module X = module Y.
>
> MODULE X/PA.6 → MODULE Y/PA.4 if there exists a module Z such that a PDR is in existence between module Y and module Z via mechanism 3 and module X has precedence over module Y.
>
> MODULE X/PA.5 for message i → MODULE Y/VPA.5 for the statement corresponding to a WAIT for message i.

An algorithm has been developed for identifying the existence of the interprocess communication mechanism in a program and the modules in a program affected by this mechanism. Interprocess communication can be identified in the software when synchronization primitives such as P and V operators or

WAIT and POST macros are utilized. It is then possible to perform a static analysis of the system to identify the modules involved in the communication. A performance dependency relationship can then be established between the modules sending the message and the modules receiving the message.

An algorithm for identifying the interprocess communication mechanism in a program will now be formally described.

3.3.1 Algorithm to Identify the Interprocess Communication Mechanism

Algorithm 3.3.1

Step 1: Construct set $P = \{(x,y) | x$ and y are modules involved in interprocess communication, where module x directly transmits a message to module $y\}$.

Step 2: For each pair of modules i in P of the form (x,y), where x is not the main program, construct $S_i = \{$modules invoking module $x\}$.

Step 3: For each module excluding the main program in S_i, construct $T_i = \{$all modules invoking modules in $S_i\}$.

Step 4: Construct $S_i' = S_i \cup T_i$. If $S_i' \neq S_i$, then set $S_i = S_i'$ and go to Step 3, otherwise go to Step 5.

Step 5: For each pair of modules i in P of the form (x,y), set $P = P \cup \{(x',y) | x' \in S_i\}$. A PDR is then defined to exist between the first and second element of each ordered pair.

3.3.2 An Example for Illustrating Algorithm 3.3.1

To illustrate Algorithm 3.3.1, let us consider the program whose R-Net is shown in Figure 12.

Step 1: $P = \{(Z,W)\}$

Step 2: $S_1 = \{Y\}$

Step 3: $T_1 = \{B\}$

Step 4: $S_1' = \{Y,B\}$

Since $S_1' \neq S$, $S_1 = \{B,Y\}$

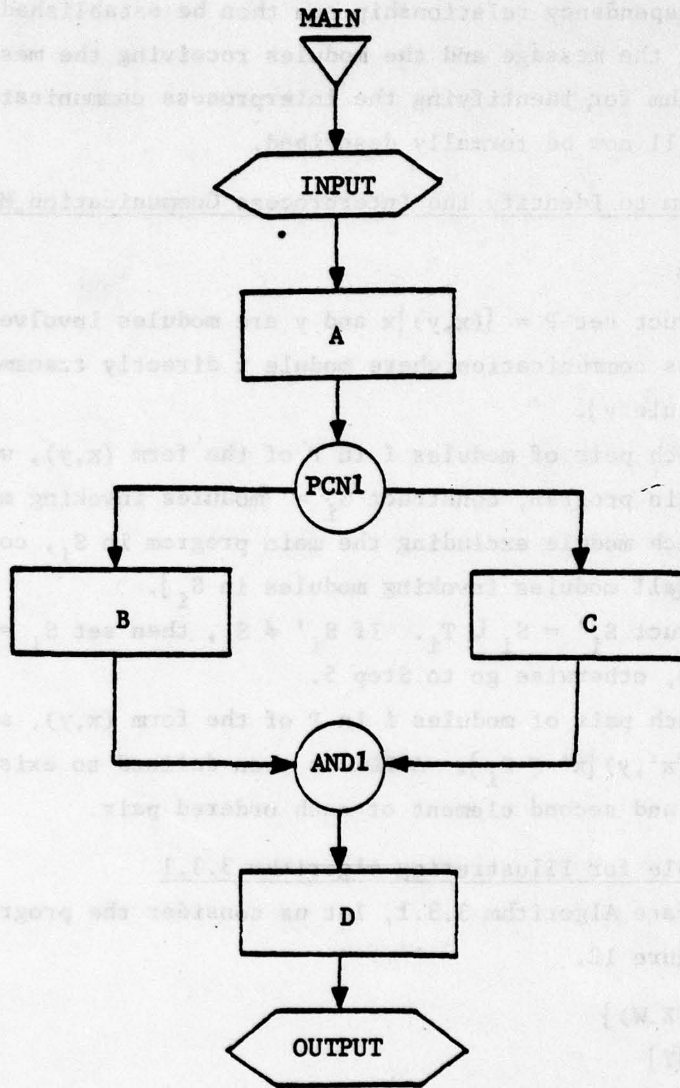
Step 3: $T_1 = \{B,MAIN\}$

Step 4: $S_1' = \{B,Y,MAIN\}$

Since $S_1' \neq S$, $S_1 = \{B,Y,MAIN\}$

Step 3: $T_1 = \{B,MAIN\}$

Step 4: $S_1' = \{B,Y,MAIN\}$



Interprocess Communication for the Program

1. Module Z sends a message to module W

Figure 12. An example for illustrating Algorithm 3.3.1.

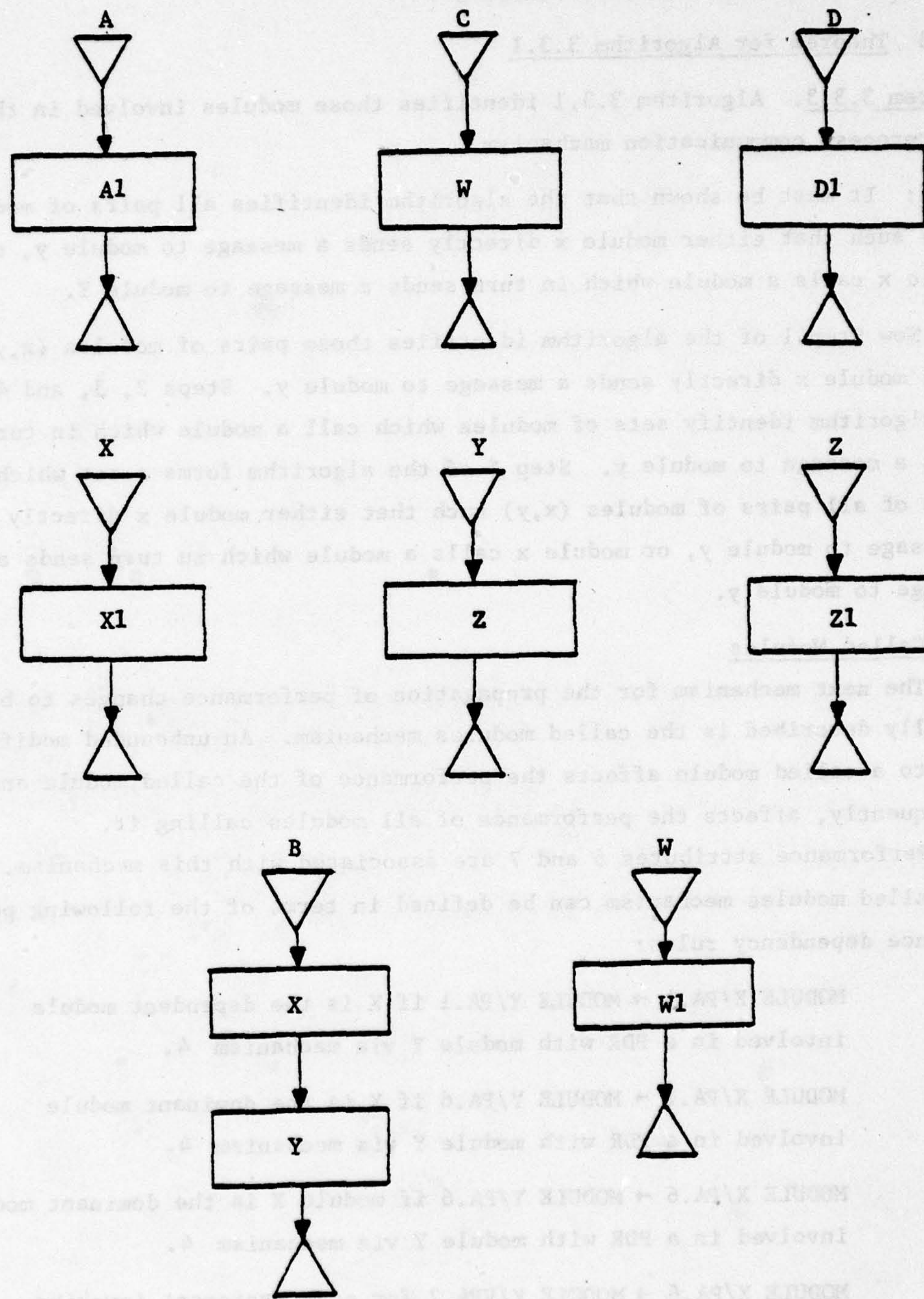


Figure 12. An example for illustrating Algorithm 3.3.1. (cont.)

Step 5: P = {(Z,W),(B,W),(Y,W),(MAIN,W)}

3.3.3 Theorem for Algorithm 3.3.1

Theorem 3.3.3. Algorithm 3.3.1 identifies those modules involved in the interprocess communication mechanism.

Proof: It must be shown that the algorithm identifies all pairs of modules (x,y) such that either module x directly sends a message to module y, or module x calls a module which in turn sends a message to module Y.

Now Step 1 of the algorithm identifies those pairs of modules (x,y), where module x directly sends a message to module y. Steps 2, 3, and 4 of the algorithm identify sets of modules which call a module which in turn sends a message to module y. Step 5 of the algorithm forms a set which consists of all pairs of modules (x,y) such that either module x directly sends a message to module y, or module x calls a module which in turn sends a message to module y.

3.4 Called Modules

The next mechanism for the propagation of performance changes to be formally described is the called modules mechanism. An unbounded modification to a called module affects the performance of the called module and, consequently, affects the performance of all modules calling it.

Performance attributes 6 and 7 are associated with this mechanism. The called modules mechanism can be defined in terms of the following performance dependency rules:

MODULE X/PA.1 → MODULE Y/PA.1 if X is the dependent module involved in a PDR with module Y via mechanism 4.

MODULE X/PA.1 → MODULE Y/PA.6 if X is the dominant module involved in a PDR with module Y via mechanism 4.

MODULE X/PA.6 → MODULE Y/PA.6 if module X is the dominant module involved in a PDR with module Y via mechanism 4.

MODULE X/PA.6 → MODULE Y/VPA.2 for each statement invoking module X in module Y.

MODULE X/PA.7 for resource i → MODULE Y/PA.7 for resource i if module X is the dominant module involved in a PDR with module Y via mechanism 4.

MODULE X/PA.7 for resource i → MODULE Y - ALPHA Z/PA.7 for resource i if module X is the dominant module involved in a PDR with module Y via mechanism 4, and module X is called from Alpha Z.

An algorithm has been developed for identifying the existence of the called modules mechanism in a program and the modules in a program affected by this mechanism. It is very easy to identify those modules in the system that are called by other modules by performing a static analysis of the system. Called modules can then be identified and performance dependency relations established between the called modules and the modules which call them.

An algorithm for identifying the called modules mechanism in a program will now be formally described.

3.4.1 Algorithm to Identify the Called Modules Mechanism

Algorithm 3.4.1

Step 1: For each module i, initialize M_i = {modules directly invoked by module i}.

Step 2: If $M_i = \emptyset$ for all M_i, terminate.

Step 3: For each non-empty M_i, construct T_i = {modules directly invoked by any module in M_i}.

Step 4: Construct $M_i' = M_i \cup T_i$. If $M_i = M_i'$ for each nonempty M_i, terminate; else set $M_i = M_i'$ and go to Step 3. A PDR is then defined to exist between each module in M_i and module i.

3.4.2 An Example to Illustrate Algorithm 3.4.1

To illustrate Algorithm 3.3.2, let us consider the program whose R-Net is shown in Figure 13.

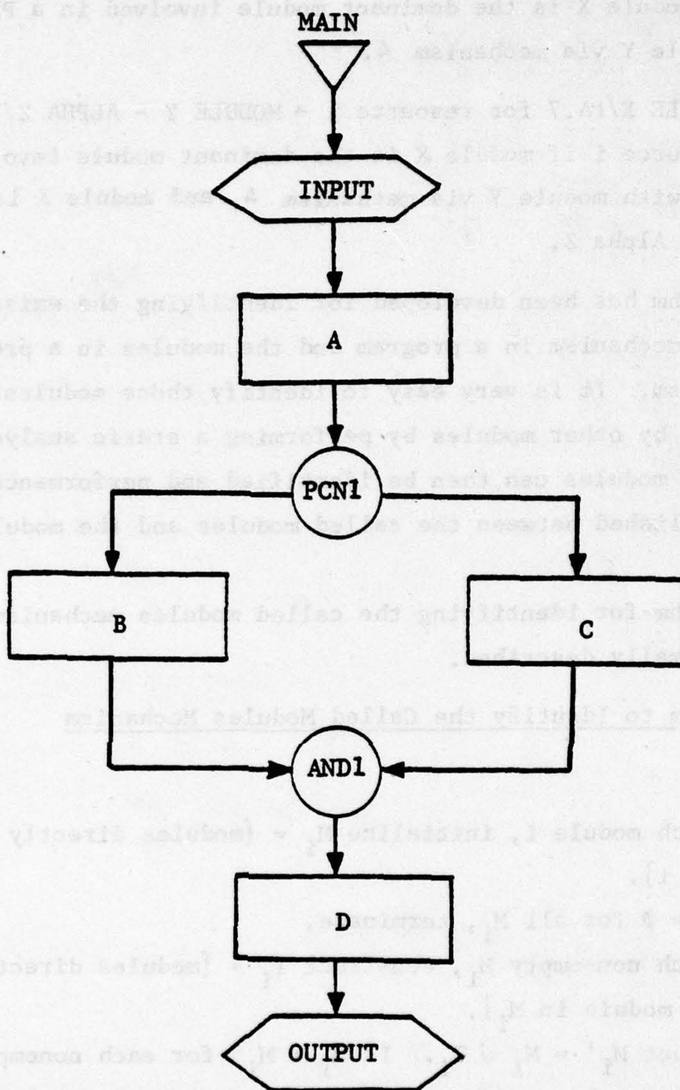


Figure 13. An example for illustrating Algorithm 3.4.1.

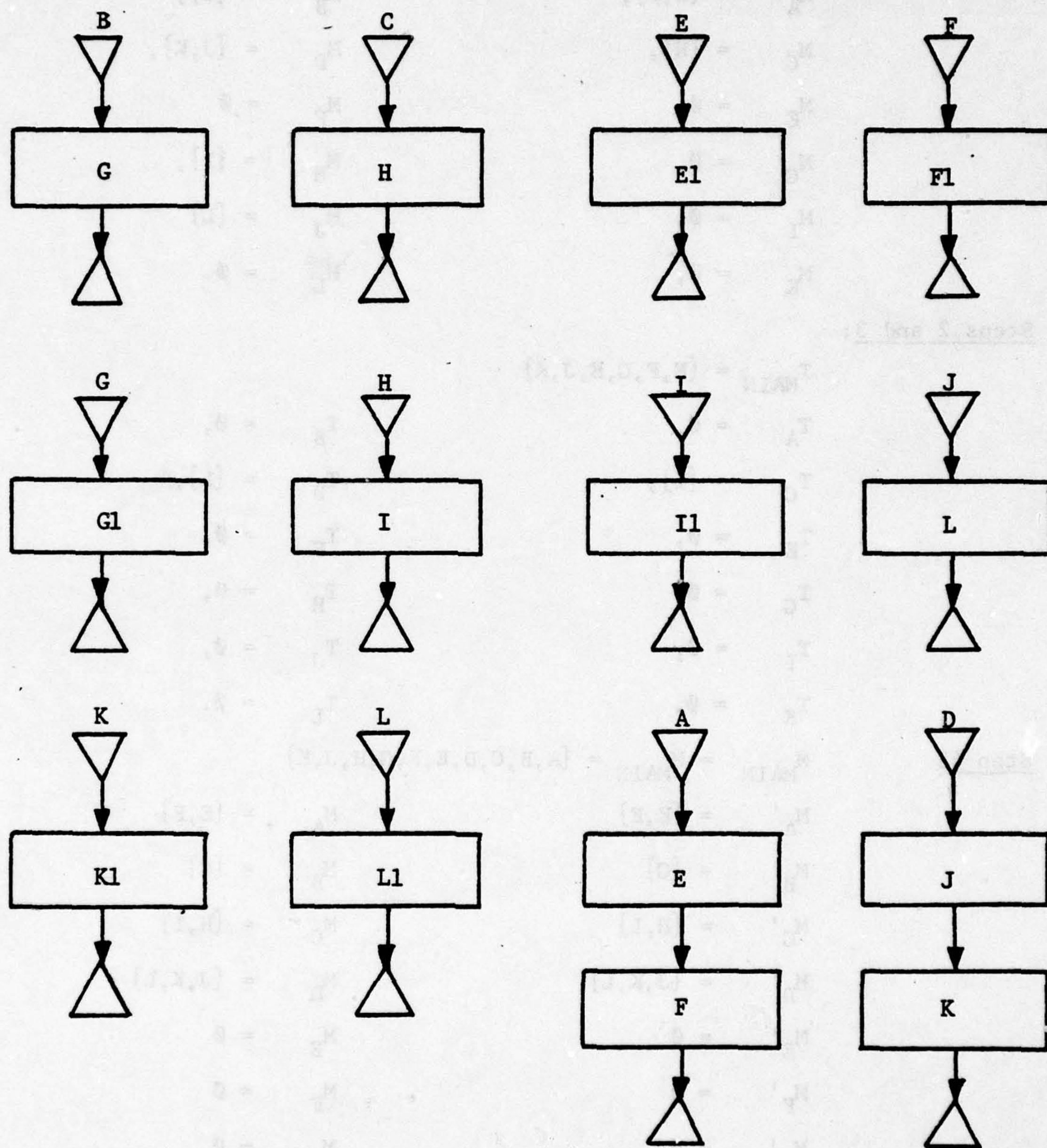


Figure 13. An example for illustrating Algorithm 3.4.1. (cont.)

Step 1: $M_{MAIN} = \{A,B,C,D\}$

$M_A = \{E,F\}$, $M_B = \{G\}$,

$M_C = \{H\}$, $M_D = \{J,K\}$,

$M_E = \emptyset$, $M_F = \emptyset$

$M_G = \emptyset$, $M_H = \{I\}$,

$M_I = \emptyset$, $M_J = \{L\}$

$M_K = \emptyset$, $M_L = \emptyset$.

Steps 2 and 3:

$T_{MAIN} = \{E,F,G,H,J,K\}$

$T_A = \emptyset$, $T_B = \emptyset$,

$T_C = \{I\}$, $T_D = \{L\}$,

$T_E = \emptyset$, $T_F = \emptyset$,

$T_G = \emptyset$, $T_H = \emptyset$,

$T_I = \emptyset$, $T_J = \emptyset$,

$T_K = \emptyset$, $T_L = \emptyset$.

Step 4: $M_{MAIN}' = M_{MAIN} = \{A,B,C,D,E,F,G,H,J,K\}$

$M_A' = \{E,F\}$ $M_A = \{E,F\}$

$M_B' = \{G\}$ $M_B = \{G\}$

$M_C' = \{H,I\}$ $M_C = \{H,I\}$

$M_D' = \{J,K,L\}$ $M_D = \{J,K,L\}$

$M_E' = \emptyset$ $M_E = \emptyset$

$M_F' = \emptyset$ $M_F = \emptyset$

$M_G' = \emptyset$ $M_G = \emptyset$

$M_H' = \{I\}$ $M_H = \{I\}$

$M_I' = \emptyset$ $M_I = \emptyset$

$M_J' = \{L\}$ $M_J = \{L\}$

$M_K' = \emptyset$ $\qquad$ $M_K = \emptyset$

$M_L' = \emptyset$ $\qquad$ $M_L = \emptyset$

<u>Step 3</u>: $\quad T_{MAIN} = \{E,F,G,H,J,K,I,L\}$

$T_A = \emptyset,$ $\qquad$ $T_B = \emptyset,$

$T_C = \{I\},$ $\qquad$ $T_D = \{L\},$

$T_E = \emptyset,$ $\qquad$ $T_F = \emptyset,$

$T_G = \emptyset,$ $\qquad$ $T_H = \emptyset,$

$T_I = \emptyset,$ $\qquad$ $T_J = \emptyset,$

$T_K = \emptyset,$ $\qquad$ $T_L = \emptyset.$

<u>Step 4</u>: $\quad M_{MAIN}' = M_{MAIN} = \{A,B,C,D,E,F,G,H,I,J,K,L\}$

$M_A' = \{E,F\},$ $\qquad$ $M_B' = \{G\},$

$M_C' = \{H,I\},$ $\qquad$ $M_D' = \{J,K,L\},$

$M_E' = \emptyset,$ $\qquad$ $M_F' = \emptyset,$

$M_G' = \emptyset,$ $\qquad$ $M_H' = \{I\},$

$M_I' = \emptyset,$ $\qquad$ $M_J' = \{L\},$

$M_K' = \emptyset,$ $\qquad$ $M_L' = \emptyset,$

<u>Step 3</u>: $\quad T_{MAIN} = \{E,F,G,H,J,K,I,L\}$

$T_A = \emptyset,$ $\qquad$ $T_B = \emptyset,$

$TC = \{I\},$ $\qquad$ $T_D = \{L\},$

$T_E = \emptyset,$ $\qquad$ $T_F = \emptyset,$

$T_G = \emptyset,$ $\qquad$ $T_H = \emptyset,$

$T_I = \emptyset,$ $\qquad$ $T_J = \emptyset,$

$T_K = \emptyset,$ $\qquad$ $T_L = \emptyset.$

Step 4: $M_{MAIN}' = \{A,B,C,D,E,F,G,H,I,J,K,L\}$

$M_A' = \{E,F\}$, $M_B' = \{G\}$,

$M_C' = \{H,I\}$, $M_D' = \{J,K,L\}$,

$M_E' = \emptyset$, $M_F' = \emptyset$,

$M_G' = \emptyset$, $M_H' = \{I\}$,

$M_I' = \emptyset$, $M_J' = \{L\}$,

$M_K' = \emptyset$, $M_L' = \emptyset$.

The following PDRs are then defined:

A ~ MAIN B ~ MAIN

C ~ MAIN D ~ MAIN

E ~ MAIN F ~ MAIN

G ~ MAIN H ~ MAIN

I ~ MAIN J ~ MAIN

K ~ MAIN L ~ MAIN

E ~ A F ~ A

G ~ B H ~ C

I ~ C J ~ D

K ~ D L ~ D

I ~ H L ~ J

3.4.3 Theorem for Algorithm 3.4.1

Theorem 3.4.3: Algorithm 3.4.1 identifies the set of modules M_i that are invoked either directly or indirectly by module i.

Proof: Let W_i = {modules invoked either directly or indirectly by module i}. It must be shown that $W_i = M_i$, where M_i is computed by Algorithm 3.4.1.

Assume that module $x \in W_i$. This implies that module x is invoked either directly or indirectly by module i. If x is invoked directly by module i, then Step 1 assures $x \in M_i$. If x is invoked indirectly by module i, then there exists a sequence of modules i, x_1, x_2,...,x_k, x such that i directly invokes x_1, x_1 directly invokes x_2,..., and x_k directly invokes x. Now, Steps 3 and 4 insure that $x_1,x_2,\ldots,x_k \in M_i$, which implies that $x \in M_i$

since x_k invokes x. Therefore, $W_i \subset M_i$.

Now assume that $x \in M_i$. This implies that x is either directly invoked by module i or invoked by a module which is indirectly invoked by module i due to the construction of M_i. This is equivalent to saying that $x \in W_i$. Therefore, $M_i \subset W_i$.

The combination of $W_i \subset M_i$ and $M_i \subset W_i$ implies that $M_i = W_i$.

3.5 Shared Data Structure

The next mechanism for the propagation of performance changes to be formally described is the shared data structures mechanism. The basic dynamic attributes contributing to performance in this area are a module's storage and retrieval times for entries in the data structures. Modifications which affect the quantity of data stored must be analyzed to determine the effect of the modification on the performance of the modules utilizing the shared data structure.

Performance attributes 8, 9, and 10 are associated with this mechanism. The shared data structures mechanism can be defined in terms of the following performance dependency rules:

DS.X/PA.9 → DS.Y/PA.8 if DS.X = DS.Y.

DS.X/PA.9 → DS.Y/PA.10 if DS.X = DS.Y.

MODULE X/PA.11 → DS.Y/PA.9 if module X stores the input in data structure Y.

DS.X/PA.8 → MODULE Y/VPA.4 for each statement referencing data structure X in module Y.

An algorithm has been developed for identifying the existence of the shared data structures mechanism in a program and the modules and data structures affected by this mechanism.

The modules manipulating shared data structures are classified into four categories based upon their utilization of the data structure. The categories are:

1. Reference entries only
2. Update entries
3. Create new entries
4. Delete old entries

It is, of course, possible for a single module to exist in more than one category.

The algorithm for identifying performance dependencies in this area is based upon the general notion that the number of entries in the data structure affects storage and retrieval times. One step of the algorithm is then the classification of the modules sharing the data structure according to the above four categories. Performance dependency relationships are then established between the modules creating and deleting entries and the other modules sharing the data structure. The relationships are valid since a modification to the modules creating or deleting entries may result in an increase in the number of entries in the data structure. This may affect storage and retrieval times, and ultimately the performance of the other modules sharing the data structure.

An algorithm for identifying the shared data structures mechanism in a program will now be formally described.

3.5.1 Algorithm to Identify the Shared Data Structures Mechanism

Algorithm 3.5.1

Step 1: Construct D = {shared data structures}.

Step 2: Construct R_i = {modules sharing data structure i}.

Step 3: Construct R_i' = {modules that create or delete entires from data structure i}. $R_i' \subset R_i$.

A PDR is then defined to exist between the modules in R_i' and those in R_i since a change in the contents of data structure i produced by a module in R_i' can affect all the modules in R_i.

3.5.2 Proof of Algorithm 3.5.1

Theorem 3.5.2: Algorithm 3.5.1 identifies those modules involved in a performance dependency relationship via the shared data mechanism.

Proof: The proof of the theorem requires proofs of the following assertions:

1. A performance dependency relationship exists between each module in R_i' and R_i.
2. All performance dependency relationships via the shared data mechanism are between modules in R_i' and R_i.

Proof of (1): Let x be an arbitrary module such that $x \in R_i'$. Let y be an arbitrary module such that $y \in R_i$. Now, $x \in R_i'$ implies by Step 3 of the algorithm that x creates or deletes entries from data structure i, and $y \in R_i$ implies module y shares data structure i. Therefore, a performance dependency relationship exists between modules x and y by definition of the shared data mechanism.

Proof of (2): Assume the contrary, i.e. there exist modules x and y in a performance dependency relationship via the shared data mechanism, but there does not exist an i such that $x \in R_i'$ and $y \in R_i$. Now, a performance dependency relationship between x and y via the shared data mechanism implies that there exist some shared data structure $i \in D$. This implies x and $y \in R_i$. Furthermore, module x must create or delete entries in data structure i for the performance dependency relationship to exist. This implies that $x \in R_i'$, which is impossible. Therefore, all performance dependency relationships via the shared data mechanism are between modules in R_i' and R_i.

3.6 Sensitivity to the Rate of Input

The next mechanism for the propagation of performance changes to be formally described is the sensitivity to the rate of input mechanism. Changes in the input rate to a process can have major repercussions in terms of its functional and performance requirements. For example, it can lead to saturation and possibly overflow of data structures involved with the processing of the input. The increased frequency of input arrivals may also lead to interruptions in processing which can lead to both functional and performance requirement violations.

Performance attribute 11 is associated with this mechanism. The sensitivity to the rate of input mechanism cannot be defined in terms of performance dependency rules as were the previous mechanisms. This is a consequence of the fact that performance attribute 11 is not affected by a change in performance of any other performance attribute, nor does it have a critical section associated with it. Instead, it is affected by a change of the program's operating environment. An environmental performance dependency relationship (EPDR) can be defined in the following manner. Let f be an attribute of the program's operating environment and let N_f be the set of modules which interfaces with f. An EPDR is defined between f and each of

the modules in N_f if a change in the attribute f will have an effect on a performance attribute for each of the modules in N_f. Thus, an EPDR is defined for each input interrupt of type i and the modules which interface with the environment to handle the input interrupts of type i.

An algorithm for identifying the sensitivity to the rate of input mechanism in a program will now be formally described.

3.6.1 Algorithm to Identify the Sensitivity to the Rate of Input Mechanism

Algorithm 3.6.1

Step 1: Identify those sets of modules, N_i, which interface with the environment to handle input interrupts of type i.

Step 2: An EPDR is then defined to exist between the program's operating environment and each module in N_i.

The validity of this algorithm is obvious.

3.7 Execution Priorities

The next mechanism for the propagation of performance changes to be formally described is the execution priorities mechanism. It is important for maintenance personnel to recognize the effect of a proposed modification in respect to the existing priorities in the system. For example, if module A has the ability to interrupt the execution of module B, then any modification affecting the execution time of module B must be carefully analyzed in order to determine if module B can still perform its designated function before being interrupted. Also, modifications to module A can affect the performance of module B since module B must wait until the execution of module A is completed before module B can complete its execution.

Performance attribute 6 is associated with this mechanism. The execution priorities mechanism can be defined in terms of the following performance dependency rule:

MODULE X/PA.6 → MODULE Y/PA.6 if module X is the dominant module involved in a PDR with module Y via mechanism 7.

An algorithm has been developed for identifying the existence of execution priorities mechanism in a program and the modules in a program affected by this mechanism. The execution priorities are set during the software

development phases. These priorities must be reflected in either the software implementation or in the accompanying documentation. An algorithm for identifying the execution priorities mechanism will now be formally described.

3.7.1 Algorithm to Identify the Execution Priorities Mechanism

Algorithm 3.7.1

Step 1: For each module, i, in the system, identify the set, I_i, to be the set of modules for which module i has interrupt priority; that is module i can interrupt the execution of any of the modules in I_i to execute.

Step 2: If $I_i = \emptyset$ for all modules, then terminate.

Step 3: Construct T_i = {modules invoked by any module in I_i} for all modules.

Step 4: For each module i, construct $I'_i = T_i \cup I_i$. If $I_i = I'_i$ for each module i, then terminate, else set $I_i = I'_i$ and go to Step 3. For each module i, establish a PDR between module i and each of the modules in I_i.

3.7.2 An example for Illustrating Algorithm 3.7.1

To illustrate Algorithm 3.7.1, let us consider the program whose R-Net is shown in Figure 14.

Step 1: $I_A = \{D,E\}$

Step 2 and 3: $T_A = \{F,G,H\}$

Step 4: $I'_A = \{D,E,F,G,H\}$ $I_A = \{D,E,F,G,H\}$

Step 3: $T_A = \{F,G,H\}$

Step 4: $I'_A = \{D,E,F,G,H\}$

The following PDRs are then defined:

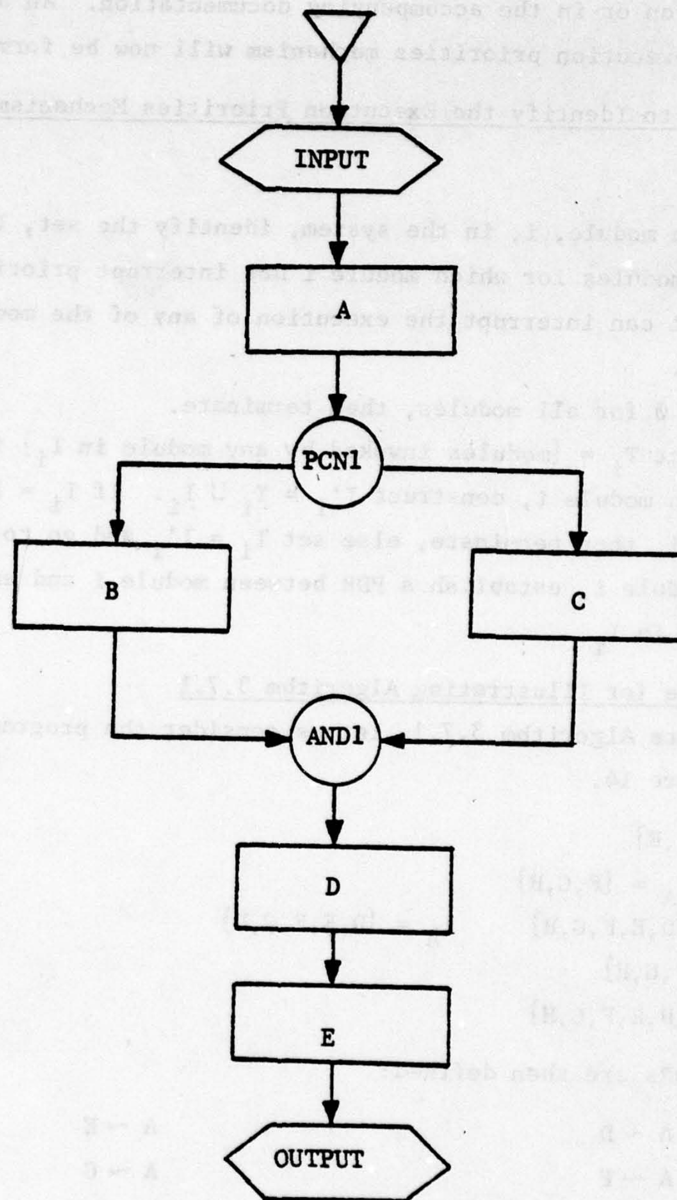
$A \sim D$ $A \sim E$

$A \sim F$ $A \sim G$

$A \sim H$

3.7.3 Theorem for Algorithm 3.7.1

Theorem 3.7.3: Algorithm 3.7.1 identifies those modules involved in a performance dependency relationship via the execution priority sensitivity mechanism.



Module A may interrupt the execution of modules D and E.

Figure 14. An example for illustrating Algorithm 3.7.1.

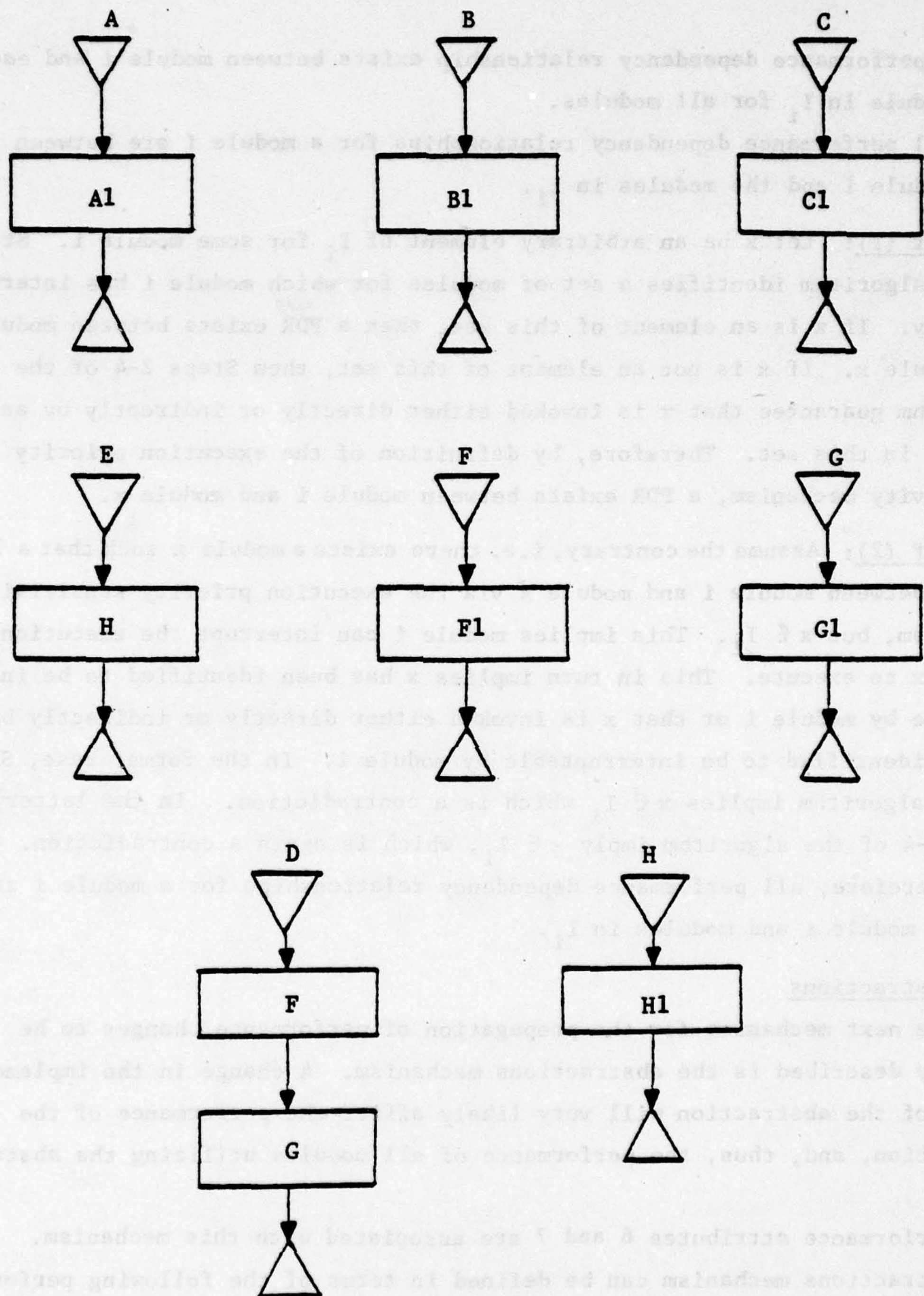


Figure 14. An example for illustrating Algorithm 3.7.1. (cont.)

<u>Proof</u>: The proof of the theorem requires the proof of the following assertions:

(1) A performance dependency relationship exists between module i and each module in I_i for all modules.

(2) All performance dependency relationships for a module i are between module i and the modules in I_i.

<u>Proof of (1)</u>: Let x be an arbitrary element of I_i for some module i. Step 1 of the algorithm identifies a set of modules for which module i has interrupt priority. If x is an element of this set, then a PDR exists between module i and module x. If x is not an element of this set, then Steps 2-4 of the algorithm guarantee that x is invoked either directly or indirectly by an element in this set. Therefore, by definition of the execution priority sensitivity mechanism, a PDR exists between module i and module x.

<u>Proof of (2)</u>: Assume the contrary, i.e. there exists a module x such that a PDR exists between module i and module x via the execution priority sensitivity mechanism, but $x \notin I_i$. This implies module i can interrupt the execution of module x to execute. This in turn implies x has been identified to be interruptable by module i or that x is invoked either directly or indirectly by a module identified to be interruptable by module i. In the former case, Step 1 of the algorithm implies $x \in I_i$ which is a contradiction. In the latter case Steps 2-4 of the algorithm imply $x \in I_i$, which is again a contradiction.

Therefore, all performance dependency relationships for a module i are between module i and modules in I_i.

3.8 <u>Abstractions</u>

The next mechanism for the propagation of performance changes to be formally described is the abstractions mechanism. A change in the implementation of the abstraction will very likely affect the performance of the abstraction, and, thus, the performance of all modules utilizing the abstraction.

Performance attributes 6 and 7 are associated with this mechanism. The abstractions mechanism can be defined in terms of the following performance dependency rules:

MODULE X/PA.1 → MODULE Y/PA.1 if X is the dependent module involved in a PDR with module Y via mechanism 8.

MODULE X/PA.1 → MODULE Y/PA.6 if X is the dominant module involved in a PDR with module Y via mechanism 8.

MODULE X/PA.6 → MODULE Y/PA.6 if module X is the dominant module involved in a PDR with module Y via mechanism 8.

MODULE X/PA.6 → MODULE Y/VPA.2 for each statement invoking abstraction X in module Y.

MODULE X/PA.7 for resource i → MODULE Y/PA.7 for resource i if module X is the dominant module involved in a PDR with module Y via mechanism 8.

MODULE X/PA.7 for resource i → MODULE Y-ALPHA Z/PA.7' for resource i if module X is the dominant module involved in a PDR with module Y via mechanism 8 and invoked by Alpha Z.

An algorithm has been developed for identifying the existence of the abstraction mechanism in a program and the modules in a program affected by this mechanism. The utilization of abstractions in a module can be easily identified by static analysis. Abstractions can be recognized in the module as subroutine calls, function calls, and macros. Performance dependency relationships can then be established between the implementations of the abstractions and the modules utilizing them.

An algorithm for identifying the abstraction mechanism in a program will now be formally described.

3.8.1 Algorithm to Identify the Abstraction Mechanism

Algorithm 3.8.1

Step 1: For each module i, initialize D_i = {abstractions directly invoked by module i},

Step 2: If $D_i = \emptyset$ for each module i, then terminate.

Step 3: For each module i, construct T_i = {abstractions directly invoked by any abstraction in D_i}.

Step 4: For each module i, construct $D'_i = T_i \cup D_i$. If $D'_i = D_i$ for each module i, then terminate, else set $D_i = D'_i$ and go to Step 3. A PDR is then defined to exist between the implementation of each abstraction in D_i and module i.

3.8.2 An Example for Illustrating Algorithm 3.8.1

To illustrate Algorithm 3.8.1, let us consider the program whose R-Net is shown in Figure 15.

Step 1: $D_A = \{1\}$

$D_B = \{2,3\}$

$D_C = \emptyset$

Steps 2 & 3: $T_A = \{2\}$

$T_B = \{4\}$

$T_C = \emptyset$

Step 4: $D'_A = \{1,2\}$ $\quad D_A = \{1,2\}$

$D'_B = \{2,3,4\}$ $\quad D_B = \{2,3,4\}$

$D'_C = \emptyset$ $\quad D_C = \emptyset$

Step 3: $T_A = \{2,4\}$

$T_B = \{4,5\}$

$T_C = \emptyset$

Step 4: $D'_A = \{1,2,4\}$ $\quad D_A = \{1,2,4\}$

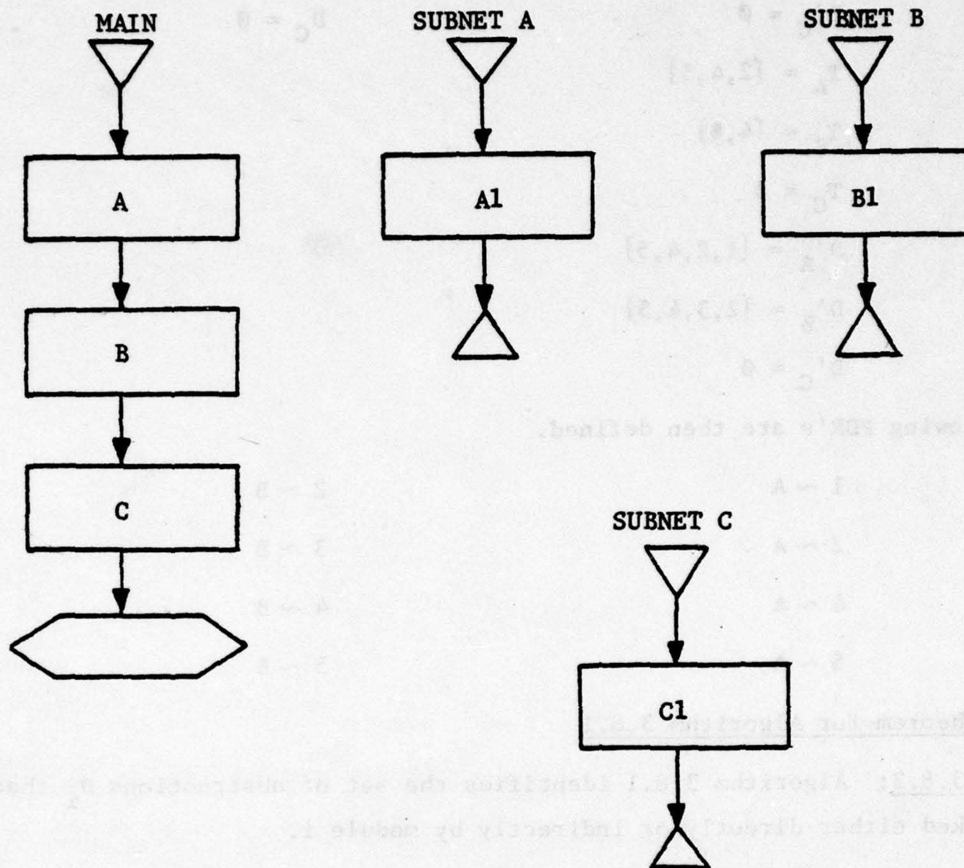
$D'_B = \{2,3,4,5\}$ $\quad D_B = \{2,3,4,5\}$

$D'_C = \emptyset$ $\quad D_C = \emptyset$

Step 3: $T_A = \{2,4,5\}$

$T_B = \{4,5\}$

$T_C = \emptyset$



Summary of static analysis results with regard to utilization of abstractions:

MODULE A invokes abstraction 1.
MODULE B invokes abstractions 2 and 3.

ABSTRACTION 1 invokes abstraction 2.
ABSTRACTION 2 invokes abstraction 4.
ABSTRACTION 4 invokes abstraction 5.

Figure 15. An example for illustrating Algorithm 3.8.1.

Step 4: $D'_A = \{1,2,4,5\}$ $D_A = \{1,2,4,5\}$

$D'_B = \{2,3,4,5\}$ $D_B = \{2,3,4,5\}$

$D'_C = \emptyset$ $D_C = \emptyset$

Step 3: $T_A = \{2,4,5\}$

$T_B = \{4,5\}$

$T_C = \emptyset$

Step 4: $D'_A = \{1,2,4,5\}$

$D'_B = \{2,3,4,5\}$

$D'_C = \emptyset$

The following PDR's are then defined.

$1 \sim A$	$2 \sim B$
$2 \sim A$	$3 \sim B$
$4 \sim A$	$4 \sim B$
$5 \sim A$	$5 \sim B$

3.8.3 Theorem for Algorithm 3.8.1

Theorem 3.8.3: Algorithm 3.8.1 identifies the set of abstractions D_i that are invoked either directly or indirectly by module i.

Proof: Let W_i = {abstractions invoked directly or indirectly by module i}. It must be shown that $W_i = D_i$, where D_i is computed by Algorithm 3.8.1.

Assume abstraction $x \in W_i$. This implies x is invoked either directly or indirectly by module i. If x is invoked directly by module i, then Step 1 assures $x \in D_i$. If x is invoked indirectly by module i, then there exists a sequence of abstractions $x_1, x_2, \ldots, x_k$, x such that i directly invokes x_1, x_1 directly invokes $x_2, \ldots, x_k$ directly invokes x. Now Steps 3 and 4 insure that $x_1, x_2, \ldots, x_k \in D_i$ implies that $x \in D_i$ since x_k invokes x. Hence, $W_i \subset D_i$.

Now assume $x \in D_i$. This implies that x is either directly invoked by module i or invoked by a module which is indirectly invoked by module i due to the construction of D_i. This is equivalent to saying that $x \in W_i$. Hence,

$M_i \subset W_i$.

Because $W_i \subset M_i$ and $M_i \subset W_i$ implies that $M_i = W_i$, this completes the proof of the theorem.

4.0 CRITICAL SECTIONS OF A PROGRAM

Since the performance attributes of a program correspond to measurements of key portions of the execution of the program, they can be affected during the maintenance process by modification to the program. A critical section of a program can be associated with some performance attributes such that if this critical section is modified, the corresponding performance attribute may be affected. For example, if the performance attribute under consideration is the execution time between when a module begins execution and when it transmits a message, the corresponding critical section is that section of code between module invocation and transmission of the message. It should be noted that a critical section for a particular performance attribute may be part of another critical section for a different performance attribute. In this case, a modification to a critical section within another critical section can affect the corresponding performance attributes of both critical sections. The relationship of performance attributes, critical software sections, and the mechanisms for the propagation of performance changes is illustrated in Figure 16 where the directed line labeled with a mechanism connecting two performance attributes indicates that a performance dependency relationship exists between the performance attributes. A directed line also connects each critical section (CS.) with its corresponding performance attribute. It is apparent from Figure 16 that a modification to CS.1 of Process A being also a modification to CS.2 implies that both PA.1 and PA.2 of Process A may be affected. Also, a change in PA.1 of Process A may affect PA.1 of Process B via mechanism 1. Thus, the modification in Process A can affect the performance of Process B.

The identification of the critical sections corresponding to the performance attributes requires algorithms whose input includes an identification of the mechanisms in existence in the program. In this section, the critical sections in a program will be defined. Associated with each of these critical sections will be a performance attribute such that if this critical section is modified, the corresponding performance attribute may

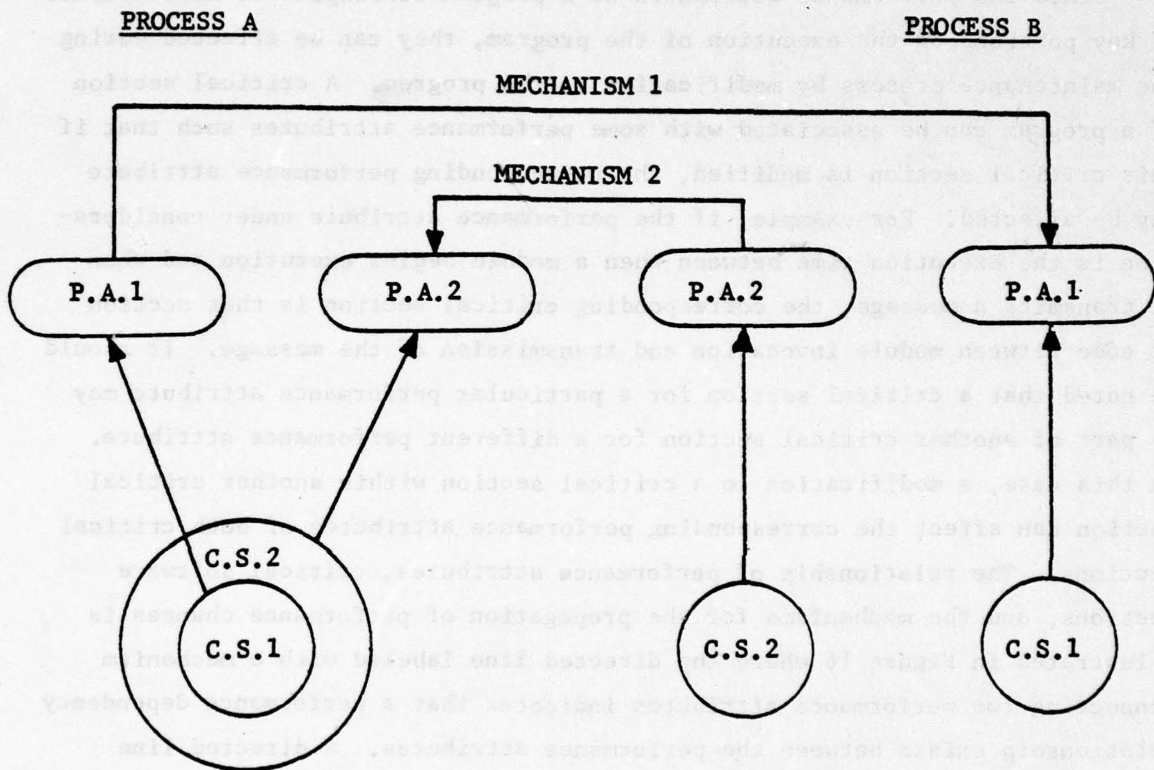


Figure 16. An example for illustrating a possible relationship among performance attributes (P.A.), critical sections (C.S.) and the mechanisms for propagation of performance changes in a program.

be affected. Also included are algorithms for identifying all critical sections of a particular type in the program. Proofs of the correctness of these algorithms are also included.

4.1 Critical Section One

Definition 4.1. A critical section of type one is defined for all modules involved in a PIR via mechanism 1 and consists of all of the blocks in the module. This critical section is associated with performance attribute 1 such that if this critical section of the module is modified, performance attribute 1 may be affected.

The following algorithm identifies all critical sections of type the program.

Algorithm 4.1. Identification of all critical sections of type

Step 1. Construct T = {all modules involved in a PIR via mechanism one}.

Step 2. For each module in T, designate all the blocks in the module to be in the critical section of type one for the module.

Theorem 4.1. Algorithm 4.1 identifies all critical sections of type one in the program.

Proof: The proof of the theorem follows directly from the definition of a critical section of type one. Step 1 of the algorithm identifies all the modules that have this critical section and Step 2 identifies all the blocks in the module to be part of the critical section. This follows exactly with the definition.

4.2 Critical Section Two

Definition 4.2. A critical section of type two is defined for all modules involved in a PIR via mechanism 2 and consists of all of the blocks in the module between its invocation and request for the resource in contention. It may also consist of all of the blocks in the module between its invocation and its call to a module which utilizes the resource in contention. A critical section of type two is associated with a performance attribute of type two for a particular resource in contention such that if this critical section is modified, the performance attribute may be affected.

The following algorithm identifies all critical sections of type two in the program.

Algorithm 4.2. Identification of all critical sections of type two.

Step 1. Select a set of modules X involved in a PIR via mechanism 2. If there are no more sets, terminate.

Step 2. For each module y in X, if y directly requests the resource i in contention among the modules in X, then designate the blocks in the module between its invocation and its request for resource i to be in the critical section of type two for the module. If the module contains multiple requests for the resource, it will have multiple critical sections.

Step 3. For each module y in X, identify Z = {all modules invoked directly or indirectly by module y}. Construct W = {modules in Z which are also involved in a PIR via mechanism 2 with resource i }.

Step 4. For each module y in X and each module v in W, designate the blocks in module y between its invocation and its call to module v to be in a critical section of type two for the module.

Step 5. Go to Step 1.

Theorem 4.2. Algorithm 4.2 identifies all critical sections of type two in the program.

Proof: The proof of the theorem follows directly from the definition of a critical section of type two. Steps 1 and 5 of the algorithm identify all of the modules involved in a PIR via mechanism 2. Step 2 of the algorithm identifies as a critical section all of the blocks in the module between its invocation and request for the resource in contention. This follows exactly with the definition. Steps 3 and 4 of the algorithm identify as a critical section those blocks in the module between its invocation and its call to a module which utilizes the resource in contention. This also follows exactly with the definition.

4.3 Critical Section Three

Definition 4.3. A critical section of type three is defined for all modules involved in a PIR via mechanism 2 which directly request the resource in contention. The critical section consists of all of the blocks in the

module between its request for the resource in contention and its release of the resource. A critical section of type three is associated with a performance attribute of type three for a particular resource in contention such that if this critical section is modified, the performance attribute may be affected.

The following algorithm identifies all critical sections of type three in the program.

Algorithm 4.3. Identification of all critical sections of type three.

Step 1. Select a set of modules X involved in a PIR via mechanism 2. If there are no more sets, terminate.

Step 2. For each module y in X, if y directly requests the resource i in contention among the modules in X, then designate the blocks in the module between its request for the resource in contention and its release of the resource to be in the critical section of type three for the module.

Step 3. Go to Step 1.

Theorem 4.3. Algorithm 4.3 identifies all critical sections of type three in the program.

Proof: The proof of the theorem follows directly from the definition of a critical section of type three. Step 1 of the algorithm identifies all of the modules involved in a PIR via mechanism 2. Step 2 of the algorithm identifies as a critical section all of the blocks in the module between its request for the resource in contention and its release of the resource. This follows exactly with the definition.

4.4 Critical Section Four

Definition 4.4. A critical section of type four is defined for all dominant modules involved in a PDR via mechanism 3 and consists of all of the blocks in the module between its invocation and its transmission of the message. It may also consist of all of the blocks in the module between its invocation and its call to a module which in turn transmits the message. A critical section of type four is associated with a performance attribute of type 5 such that if this critical section is modified, the performance attribute may be affected.

The following algorithm identifies all critical sections of type four in the program.

Algorithm 4.4. Identification of all critical sections of type four.

Step 1. Select a pair of modules X involved in a PDR via mechanism 3. If there are no more pairs, terminate.

Step 2. For the dominant module y in X, if y directly transmits the message to the other module y' in X, then designate the blocks in the module between its invocation and its transmission of the message tc be in the critical section of type four for the module.

Step 3. For the dominant module y in X, identify Z = {all modules invoked directly or indirectly by y}. Construct W = {modules in Z which are also in a PDR with module y' via mechanism 3}.

Step 4. For the dominant module y in X and each module v in W, designate the blocks in module y between its invocation and its call to module v to be in a critical section of type four for the module.

Step 5. Go to Step 1.

Theorem 4.4. Algorithm 4.4 identifies all critical sections of type four in the program.

Proof: The proof of the theorem follows directly from the definition of a critical section of type four. Steps 1 and 5 of the algorithm identify all the pairs of modules involved in a PDR via mechanism 3. Step 2 of the algorithm identifies as a critical section all of the blocks in the dominant module in the pair between its invocation and its transmission of the message. This follows exactly with the definition. Steps 3 and 4 of the algorithm identify as a critical section those blocks in the dominant module in the pair between its invocation and its call to a module which in turn transmits the message. This also follows exactly with the definition.

4.5 Critical Section Five

Definition 4.5. A critical section of type five is defined for all dominant modules involved in a PDR via mechanism 4, 7, or 8 and consists of all the blocks in the module. A critical section of type five is associated with a performance attribute of type six such that if this critical section is modified, the performance attribute may be affected.

The following algorithm identifies all critical sections of type five in the program.

Algorithm 4.5. Identification of all critical sections of type five.

Step 1. Select a pair of modules X involved in a PDR via mechanism 4, 7, or 8. If there are no more pairs, terminate.

Step 2. For the dominant module y in X, designate all of the blocks in the module to be in the critical section of type five for the module.

Step 3. Go to Step 1.

Theorem 4.5. Algorithm 4.5 identifies all critical sections of type five in the program.

Proof: The proof of the theorem follows directly from the definition of a critical section of type five. Steps 1 and 3 of the algorithm identify all of the pairs of modules involved in a PDR via mechanism 4, 7, or 8. Step 2 of the algorithm identifies all of the blocks in the dominant module in the pair as a critical section. This follows exactly with the definition.

4.6 Critical Section Six

Definition 4.6. A critical section of type six is defined for all dominant modules involved in a PDR via mechanism 4 or 8 and consists of all of the blocks in the module between its request for a resource and its release of the resource. If the module utilizes more than one resource or contains multiple requests for the same resource, it will have multiple critical sections. A critical section of type six is associated with a performance attribute of type seven for a particular resource such that if this critical section is modified, the performance attribute may be affected.

The following algorithm identifies all critical sections of type six in the program.

Algorithm 4.6. Identification of all critical sections of type six in the program.

Step 1. Select a pair of modules X involved in a PDR via mechanism 4 or 8. If there are no more pairs, terminate.

Step 2. For the dominant module y in X, if y directly requests a resource i, then designate the blocks in the module between its request for resource i and its release of resource i to be in the critical section of type six for the module.

Step 3. Go to Step 1.

Theorem 4.6. Algorithm 4.6 identifies all critical sections of type six in the program.

Proof: The proof of the theorem follows directly from the definition of a critical section of type six. Steps 1 and 3 of the algorithm identify all of the pairs of modules involved in a PDR via mechanism 4 or 8. Step 2 of the algorithm identifies as a critical section all of the blocks in the dominant module in the pair between its request for a particular resource and its release of the resource. This follows exactly with the definition.

4.7 Critical Section Seven

Definition 4.7. A critical section of type seven is defined for all modules containing a dependent iterative structure and consists of the set of variables involved in the iterative structure, excluding the index variable. Each dependent iterative structure has its own critical section. A critical section of type seven is associated with a performance attribute of type 12 such that if this critical section is modified, the performance attribute may be affected.

The following algorithm identifies all critical sections of type seven in the program.

Algorithm 4.7. Identification of all critical sections of type seven.

Step 1. Examine each module in the program and identify the dependent iterative structures in the module.

Step 2. For each dependent iterative structure in each module, designate the set of variables involved in the iterative structure, excluding the index variable as a critical section of type seven for the module.

Theorem 4.7. Algorithm 4.7 identifies all critical sections of type seven in the program.

Proof: The proof of the theorem follows trivially from the definition of a critical section of type 7.

5.0 DEPENDENCY RELATIONSHIPS BETWEEN VIRTUAL PERFORMANCE ATTRIBUTES AND PERFORMANCE ATTRIBUTES

In this section, the performance dependency relationships in existence between the virtual performance attributes and the performance attributes presented in Section 2 will be identified. These performance dependency relationships are utilized in the determination of which performance attributes are affected when a virtual performance attribute is affected. For example, in Section 2 a performance dependency relationship in existence between virtual performance attribute 2 and a performance attribute was described. In that example, the effect of changing performance attribute 6 of module A in Figure 4 was examined. The result of the change affected virtual performance attribute 2 of program X. Now since a performance dependency relationship exists between virtual performance attribute 2 and performance attribute 5 of program X, the actual effect of changing performance attribute 6 of module A is a change in performance attribute 5 of program X.

In this section, for each of the virtual performance attribute types presented in Section 2 performance dependency relationship between the virtual performance attribute for a module and the performance attributes for the module will be defined. An algorithm for each of the virtual performance attribute types will be presented to identify the performance attributes involved in the performance dependency relationship with the virtual performance attribute such that if the virtual performance attribute is affected, the performance attributes are also affected. Proofs of the correctness of these algorithms will also be provided.

5.1 Dependency Relationships for Virtual Performance Attribute of Type One

Definition 5.1. A performance dependency relationship is defined to exist between a virtual performance attribute of type one for a module and a set of performance attributes for the module that corresponds to the critical sections of code which must wait due to the denial of the contended resource

associated with the virtual performance attribute of type one.

Algorithm 5.1. Identification of the dependency relationships in existence in a module between a virtual performance attribute of type one for the module and the performance attributes for the module.

Step 1. Identify the critical sections for the module which contain the request for the resource in contention associated with the virtual performance attribute of type one.

Step 2. Identify the performance attributes for the modules which are asso- with these critical sections.

Step 3. Establish a dependency relationship between the virtual performance attribute of type one and these performance attributes such that if the virtual performance attribute is affected, it will also affect these performance attributes.

Theorem 5.1. Algorithm 5.1 identifies all of the dependency relationships in existence in a module between a virtual performance attribute of type one for the module and the performance attributes for the module.

Proof. The proof of the theorem follows directly from the definition of a performance dependency relationship between a virtual performance attribute of type one for a module and the performance attributes for the module. Step 1 identifies the critical sections of the competing module which must wait due to denial of a particular contended resource. Step 2 simply identifies the performance attributes associated with these critical sections. Step 3 then establishes a dependency relationship according to the definition of a performance dependency relationship between a virtual performance attribute of type one for a module and the performance attributes for the module.

5.2 Dependency Relationships for Virtual Performance Attribute of Type Two

Definition 5.2. A performance dependency relationship is defined to exist between a virtual performance attribute of type two for a module and a set of performance attributes for the module that corresponds to the critical sections of code containing the invocation to the module or abstraction associated with the virtual performance attribute of type two.

Algorithm 5.2. Identification of the dependency relationships in existence in a module between a virtual performance attribute of type two for the module and the performance attributes for the module.

Step 1. Identify the critical sections for the module which contain the statement which invokes the module or abstraction which is associated with the virtual performance attribute of type two.

Step 2. Identify the performance attributes for the modules which are associated with these critical sections.

Step 3. Establish a dependency relationship between the virtual performance attribute of type two and these performance attributes such that if the virtual performance attribute is affected, it will also affect these performance attributes.

Theorem 5.2. Algorithm 5.2 identifies all of the dependency relationships in existence in a module between a virtual performance attribute of type two for the module and the performance attributes for the module.

Proof: The proof of the theorem follows directly from the definition of a performance dependency relationship between a virtual performance attribute of type two for a module and the performance attributes for the module. Step 1 identifies the critical sections of the module which contain an invocation to a module or the utilization of an abstraction. Step 2 simply identifies the performance attributes associated with these critical sections. Step 3 then establishes a performance dependency relationship according to the definition of a performance dependency relationship between a virtual performance attribute of type two for a module and the performance attributes for the module.

5.3 Dependency Relationships for Virtual Performance Attribute of Type Three

Definition 5.3. A performance dependency relationship is defined to exist between a virtual performance attribute of type three for a module and a set of performance attributes for the module that corresponds to the critical sections of code which contain the storage or retrieval request of an entry from a data structure associated with the virtual performance attribute of type three.

Algorithm 5.3. Identification of the dependency relationships in existence between a virtual performance attribute of type three for the module and the performance attributes for the module.

Step 1. Identify the critical sections for the module which contain the iterated code controlled by the dependent iterative structure which is associated with the virtual performance attribute of type three.

Step 2. Identify the performance attributes for the module which are associated with these critical sections.

Step 3. Establish a dependency relationship between the virtual performance attribute of type three and these performance attributes such that if the virtual performance attribute is affected, it will also affect these performance attributes.

Theorem 5.3. Algorithm 5.3 identifies all of the dependency relationships in existence in a module between a virtual performance attribute of type three for the module and the performance attributes for the module.

Proof: The proof of the theorem follows directly from the definition of a performance dependency relationship between a virtual performance attribute of type three for a module and the performance attributes for the module. Step 1 identifies the critical sections of the module which contain the iterated code controlled by the dependent iterative structure. Step 2 simply identifies the performance attributes associated with these critical sections. Step 3 then establishes a performance dependency relationship according to the definition of a performance dependency relationship between a virtual performance attribute of type three for a module and the performance attributes for the module.

5.4 Dependency relationships for Virtual Performance Attribute of Type Four

Definition 5.4. A performance dependency relationship is defined to exist between a virtual performance attribute of type four for a module and a set of performance attributes for the module that corresponds to the critical sections of code which contain the storage or retrieval request of an entry from a data structure associated with the virtual performance attribute of type four.

Algorithm 5.4. Identification of the dependency relationships in existence in a module between a virtual performance attribute of type four for the module and the performance attributes for the module.

Step 1. Identify the critical sections for the module which contain the reference to the data structure which is associated with the virtual performance attribute of type four.

Step 2. Identify the performance attributes for the module which are associated with these critical sections.

Step 3. Establish a dependency relationship between the virtual performance attribute of type four and these performance attributes such that if the virtual performance attribute is affected, it will also affect these performance attributes.

Theorem 5.4. Algorithm 5.4 identifies all of the dependency relationships in existence in a module between a virtual performance attribute of type four for the module and the performance attributes for the module.

Proof: The proof the theorem follows directly from the definition of a performance dependency relationship between a virtual performance attribute of type four for a module and the performance attributes for the module. Step 1 identifies the critical sections of code containing references to data structures. Step 2 simply identifies the performance attributes associated with these critical sections. Step 3 then establishes a performance dependency relationship according to the definition of a performance dependency relationship between a virtual performance attribute of type four for a module and the performance attributes for the module.

5.5 Dependency Relationships for Virtual Performance Attribute of Type Five

Definition 5.5. A performance dependency relationship is defined to exist between a virtual performance attribute of type five for a module and a set of performance attributes for the module that corresponds to the critical sections of code which must wait for the message to be transmitted from another module which is associated with the virtual performance attribute of type five.

Algorithm 5.5. Identification of the dependency relationships in existence in a module between a virtual performance attribute of type five for the module and the performance attributes for the module.

Step 1. Identify the critical sections for the module which contain the WAIT statement for the message associated with the virtual performance attribute of type five.

Step 2. Identify the performance attributes for the module which are associated with these critical sections.

Step 3. Establish a dependency relationship between the virtual performance attribute of type five and these performance attributes such that if the virtual performance attribute is affected, it will also affect these performance attributes.

Theorem 5.5. Algorithm 5.5 identifies all of the dependency relationships in existence in a module between a virtual performance attribute of type five and the performance attributes for the module.

Proof: The proof of the theorem follows directly from the definition of a performance dependency relationship between a virtual performance attribute of type five for a module and the performance attributes for the module. Step 1 identifies the critical sections of code which must wait for a message to be transmitted from another module. Step 2 simply identifies the performance attributes associated with these critical sections. Step 3 then establishes a performance dependency relationship according to the definition of a performance dependency relationship between a virtual performance attribute of type five for a module and the performance attributes for the module.

6.0 PERFORMANCE DEPENDENCY RELATIONSHIP RULES

In this section, all of the performance dependency relationship rules utilized in the formal description of the mechanisms for the propagation of performance changes in Section 3 will be compiled. This list of performance dependency relationship rules will be completed with rules which were not utilized in the description of the mechanisms.

The performance dependency relationships among the performance attributes in the program are described according to the following set of rules. The rules are of the format:

MODULE X/PA.Y → MODULE Z/PA.W CONDITION

and are interpreted as follows:

A change in performance attribute Y of module X may affect performance attribute W of module Z if the condition is satisfied. A variation of this format is the replacement of MODULE with DS. which represents data structure. DS. X/PA.Y is then interpreted as the Yth performance attribute of data structure X.

The rules for describing performance dependency relationships among the performance attributes in the program are the following:

MODULE X/PA.1 → MODULE Y/PA.1 if X is the dependent module involved in a PDR with module Y via mechanism 1, 4, or 8.

MODULE X/PA.1 → MODULE Y/PA.4 if X is involved in a PDR with module Y via mechanism 1.

MODULE X/PA.1 → MODULE Y/PA.6 if X is the dominant module involved in a PDR with module Y via mechanism 4, 8, or 1.

MODULE X/PA.2 for resource i → MODULE Y/PA.2 for resource i if module X is involved in a PIR with module Y via mechanism 2.

MODULE X/PA.2 for resource i → MODULE Y/PA.3 for resource i if module X = module Y or module X is involved in a PIR with module Y via mechanism 2.

MODULE X/PA.2 for resource i → MODULE Y/PA.4 if module X is involved in a PIR with module Y via mechanism 2.

MODULE X/PA.2 for resource i → MODULE Y/PA.7 for resource i if module X = module Y.

MODULE X/PA.3 for resource i → MODULE Y/PA.2 for resource i if module X is involved in a PIR with module Y via mechanism 2.

MODULE X/PA.3 for resource i → MODULE Y/PA.3 for resource i if module X is involved in a PIR with module Y via mechanism 2.

MODULE X/PA.3 for resource i → MODULE Y/PA.4 if module X is involved in a PIR with module Y via mechanism 2.

MODULE X/PA.3 for resource i → MODULE Y/PA.7 for resource i if module X = module Y.

MODULE X/PA.4 → MODULE Y/PA.2 for resource i if module X = module Y.

MODULE X/PA.4 → MODULE Y/PA.3 for resource i if module X = module Y.

MODULE X/PA.4 → MODULE Y/PA.4 if there exists a module Z such that a PDR is in existence between module Y and module Z via mechanism 2 or 3 and module X has precedence over module Y.

MODULE X/PA.4 → MODULE Y/PA.5 for message i if module X = module Y.

MODULE X/PA.6 → MODULE Y/PA.4 if there exists a module Z such that a PDR is in existence between module Y and module Z via mechanism 2 or 3 and module X has precedence over module Y.

MODULE X/PA.6 → MODULE Y/PA.6 if module X is the dominant module involved in a PDR with module Y via mechanism 4, 7, or 8.

MODULE X/PA.7 for resource i → MODULE Y/PA.7 for resource i if module X is the dominant module involved in a PDR with module Y via mechanism 4 or 8.

MODULE X/PA.7 for resource i → MODULE Y-ALPHA Z/PA.7' for resource i if module X is the dominant module involved in a PDR with module Y via mechanism 4 and the call to module X is in Alpha z.

DS.X/PA.9 → DS.Y/PA.8 if DS.X = DS.Y

DS.X/PA.9 → DS.Y/PA.10 if DS.X = DS.Y

MODULE X/PA.11 → DS.Y/PA.9 if module X stores the input in data structure Y.

The performance dependency relationships in existence between the performance attributes and the virtual performance attributes in the program are described according to the following set of rules. The rules are of the format:

MODULE X/PA.Y → MODULE Z/VPA.W CONDITION

and are interpreted as follows:

A change in performance attribute Y of module X may affect virtual performance attribute W of module Z if the condition is satisfied. The rules for describing performance dependency relationships between the performance attributes and the virtual performance attributes in the program are the following:

MODULE X/PA.2 for resource i → MODULE Y/VPA.1 for the request for resource i if module X is involved in a PIR with module Y via mechanism 2.

MODULE X/PA.3 for resource i → MODULE Y/VPA.1 for the request for resource i if module X is involved in a PIR with module Y via mechanism 2.

MODULE X/PA.6 → MODULE Y/VPA.2 for each statement invoking module X in module Y.

DS.X/PA.8 → MODULE Y/VPA.4 for each statement referencing data structure X in module Y.

MODULE X/PA.5 for message i → MODULE Y/VPA.5 for the statement corresponding to a WAIT for message i.

One additional performance dependency relationship can be defined between a performance attribute and a virtual performance attribute that was not defined in Section 3. The performance dependency relationship is described according to the following rule:

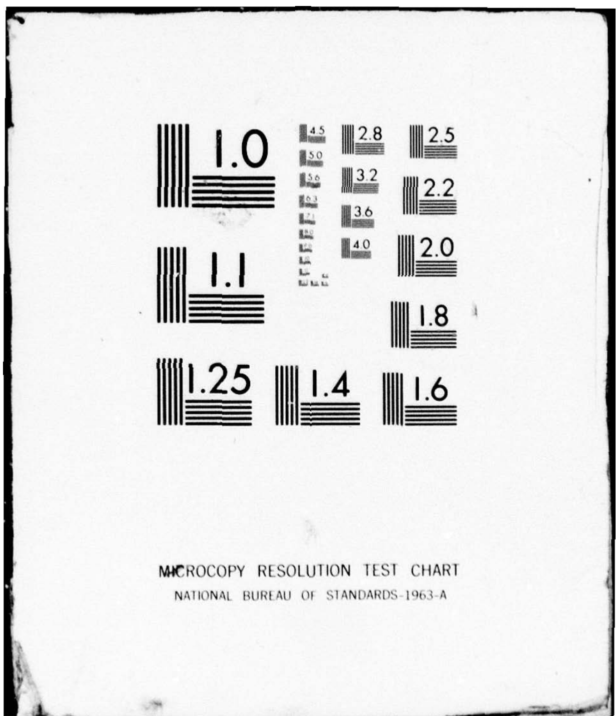
MODULE X/PA.12 for dependent iterative structure i → MODULE Y/VPA.3 for dependent iterative structure i if module X = module Y.

This performance dependency relationship rule was not defined in Section 3 since there are not any mechanisms for the propagation of performance changes defined in terms of this performance dependency relationship. This is a consequence of the fact that performance attribute 12 is not related to the eight mechanisms for the propagation of performance changes as discussed in Section 2.

AD-A084 351 NORTHWESTERN UNIV EVANSTON IL DEPT OF ELECTRICAL ENG--ETC F/G 9/2
PERFORMANCE RIPPLE EFFECT ANALYSIS FOR LARGE-SCALE SOFTWARE MAI--ETC(U)
MAR 80 S S YAU, J S COLLOFELLO F30602-76-C-0397

UNCLASSIFIED RADC -TR-80-55 NL

2 OF 2
AD-
A08435I


1.0
4.5
5.0
5.6
6.3
2.8
3.2
3.6
4.0
2.5
2.2
2.0
1.1
1.8
1.25
1.4
1.6
MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

7.0 RIPPLE EFFECT OF PERFORMANCE CHANGES

The maintainability of a software system is a measure of the ease of performing maintenance to the software system, and all maintenance activities require modifications to the system. The effect of a modification to a software system may not be local to the location of the modification, but may also affect other portions of the system. This means a ripple effect existing from the location of the modification to the other parts of the system that are affected by the modification. One aspect of this ripple effect is logical or functional in nature, and another concerns the performance of the system.

The concept of a performance change ripple effect as a consequence of software modification is analyzable by examination of the relationship of performance attributes, virtual performance attributes, critical sections and the mechanisms for the propagation of performance changes. Performance dependency relationships among performance attributes and between performance attributes and virtual performance attributes were described in terms of the performance dependency rules presented in Section 6. The performance dependency relationships between virtual performance attributes and performance attributes were described in Section 5. With these performance dependency relationships it is possible to determine the ripple effect of performance changes as a consequence of program modification.

This ripple effect of performance changes is computed by first identifying the corresponding performance attributes affected by modification of a critical section. A change in these performance attributes may then ripple, i.e., affect other performance attributes or virtual performance attributes according to the applicable performance dependency relationship rules. If a virtual performance attribute is affected, the corresponding performance attributes which are affected can be identified by the algorithms in Section 5. These affected performance attributes may then affect still other performance attributes and virtual performance attributes which accounts for the ripple effect of performance changes.

For example, consider the three modules of a program presented in Figure 17. If a modification to module X occurs affecting performance attribute 6, there is a ripple effect of performance changes affecting all three modules. The change in performance attribute 6 will affect virtual performance attribute 2 of module Y according to the performance dependency rule

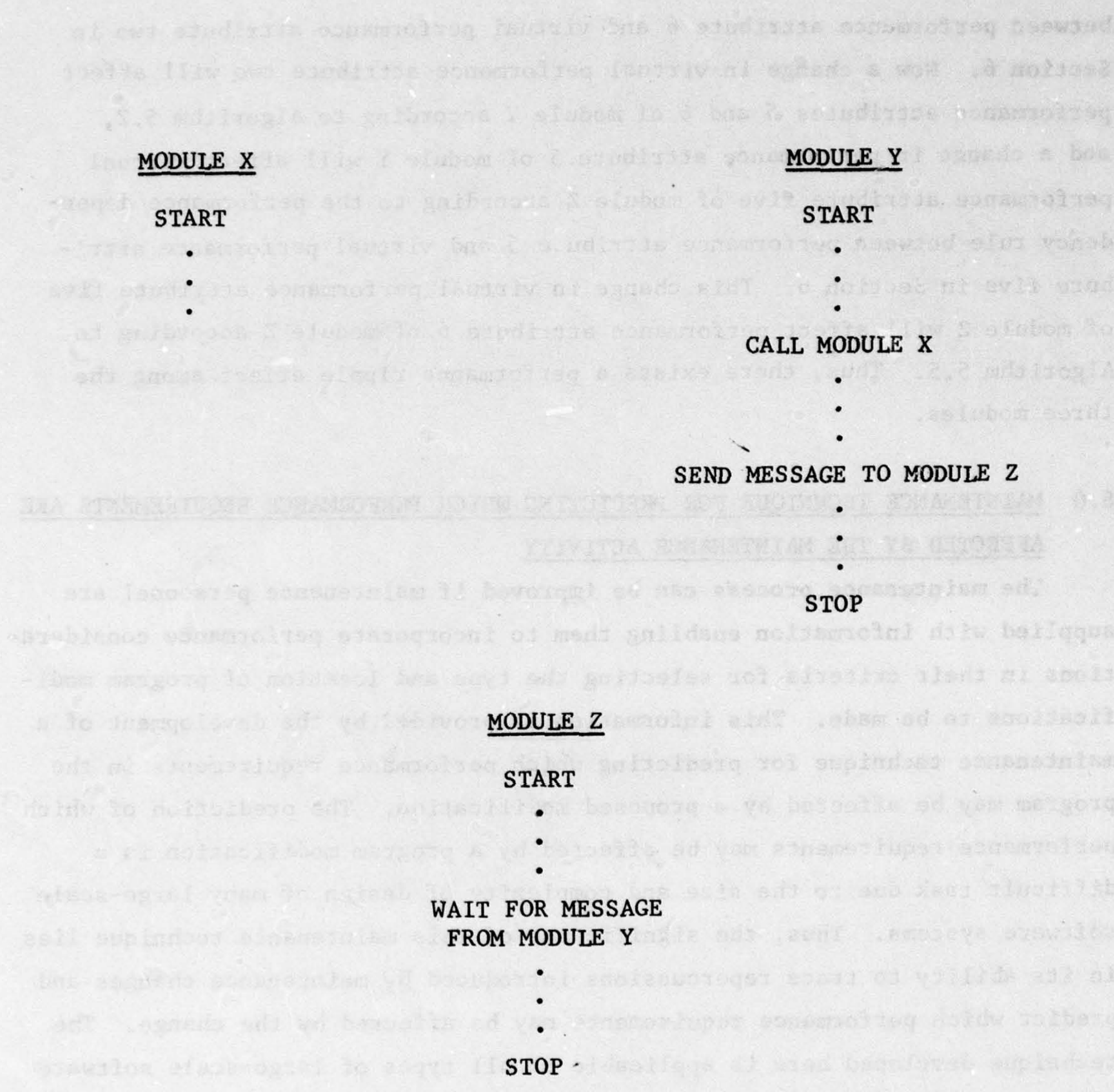


Figure 17. An example to illustrate the ripple effect of performance changes.

between performance attribute 6 and virtual performance attribute two in Section 6. Now a change in virtual performance attribute two will affect performance attributes 5 and 6 of module Y according to Algorithm 5.2, and a change in performance attribute 5 of module Y will affect virtual performance attribute five of module Z according to the performance dependency rule between performance attribute 5 and virtual performance attribute five in Section 6. This change in virtual performance attribute five of module Z will affect performance attribute 6 of module Z according to Algorithm 5.5. Thus, there exists a performance ripple effect among the three modules.

8.0 MAINTENANCE TECHNIQUE FOR PREDICTING WHICH PERFORMANCE REQUIREMENTS ARE AFFECTED BY THE MAINTENANCE ACTIVITY

The maintenance process can be improved if maintenance personnel are supplied with information enabling them to incorporate performance considerations in their criteria for selecting the type and location of program modifications to be made. This information is provided by the development of a maintenance technique for predicting which performance requirements in the program may be affected by a proposed modification. The prediction of which performance requirements may be affected by a program modification is a difficult task due to the size and complexity of design of many large-scale software systems. Thus, the significance of this maintenance technique lies in its ability to trace repercussions introduced by maintenance changes and predict which performance requirements may be affected by the change. The technique developed here is applicable to all types of large-scale software systems possessing performance requirements including multiprocessing systems.

In this section, a maintenance technique for predicting which performance requirements are affected by a proposed program modification will be formally described. Section 8.1 contains some key algorithms and guidelines composing this maintenance technique. Section 8.2 contains the complete description of the maintenance technique. Section 8.3 contains a proof of the correctness of the algorithms composing the maintenance technique. Finally, Section 8.4 provides an application of the maintenance technique to the retesting phase of the maintenance process.

8.1 Key Algorithms and Guidelines Composing the Maintenance Technique

In this section some of the key algorithms composing the maintenance technique will be formally described. Proofs of the correctness of these algorithms will also be provided. Some guidelines for the decomposition of performance requirements into performance attributes will also be discussed.

8.1.1 Guidelines for the Decomposition of Performance Requirements into Performance Attributes

An important step of the maintenance technique to be described in Section 8.2 is the decomposition of the program's performance requirements into the performance attributes for the program. In our previous report a few approaches to the statement of performance requirements were reviewed [17]. The current approach in software engineering to the statement of performance requirements in terms of flows through the system was also described. This approach utilizes R-Nets, which are also defined in our earlier report [17].

In this section, the decomposition of performance requirements into performance attributes will be investigated. The decomposition of a performance requirement quantitatively into the effect of its corresponding performance attributes is a very complex task which is not attempted in this technique. Instead, the decomposition is qualitative in nature, i.e., performance attributes which contribute to the preservation or violation of performance requirements without consideration of their relative magnitude towards the performance requirements are identified. This simplification is justified because this maintenance technique attempts to identify performance requirements which may be violated due to the maintenance effort, and does not attempt to analytically confirm whether or not a performance requirement is actually violated.

Since the current trend in stating performance requirements is at the R-Net level, in this section the emphasis will be on the decomposition of performance requirements stated at the R-Net level into performance attributes. Two general categories of performance requirements will be investigated. The first category concerns the performance requirement that a certain task be performed in a specified execution time. The second category concerns the performance requirement that this task be performed with resource

utilization restrictions. Now these performance requirements are testable if they are stated in terms of the validation points identified on the R-Net. The guidelines for the decomposition of performance requirements into performance attributes are based upon the assumption that these performance requirements are stated in terms of validation points on the R-Nets. Furthermore, it is assumed that the R-Nets are refined until each "Alpha" in the graph corresponds to a module or a segment of code in a module.

The decomposition of the performance requirements into the performance attributes can then be accomplished in the following way:

Step 1. Analyze each performance requirement. If the performance requirement states that the program executes in a specified time between two validation points, then identify the modules or segments of code corresponding to the "Alphas" on the R-Net between the validation points. For each of these modules, designate PA.6 to be a performance attribute for the module.

For each of the segments of code identified, designate PA.6' to be a performance attribute for the segment. Also, designate critical section 5' to contain all of the blocks in the segment of code corresponding to the "Alpha" on the R-Net. PA.6' is associated with CS.5' such that if CS.5' is modified, then PA.6' may be affected.

Decompose the performance requirement into performance attributes of type 6 and 6' for these Alphas.

Step 2. If the performance requirement states that the program is executed with resource utilization restrictions between two validation points, then identify the modules corresponding to the "Alphas" on the R-Net between the validation points. For each of these modules, designate PA.7 for the resource being restricted to be a performance attribute for the module. For each of the segments of code identified, designate PA.7' to be a performance attribute for the segment. Decompose the performance requirement into these performance attributes 7 and 7'.

As requirement statement languages continue to develop, it appears feasible that the decomposition of performance requirements into performance attributes may be accomplished automatically. If the performance requirements

are not stated at the R-Net level, then the decomposition of the performance requirements into the performance attributes is more complex. The decomposition requires an analysis of the modules whose execution is governed by the performance requirements. Appropriate performance attributes for these modules must then be identified. This process will be more difficult to automate than the process of decomposing performance requirements stated at the R-Net level into performance attributes. Thus, although this maintenance technique is flexible enough to support the maintenance activity of all software systems, the ideal situation occurs when performance requirements are stated at the R-Net level.

Figure 18 is an expansion of Figure 16 which includes a description of the relationship of performance requirements, performance attributes, critical sections, and the mechanisms for the propagation of performance changes in a program. In this figure, the directed line labeled with a mechanism connecting two performance attributes indicates a performance dependency relationship exists between the performance attributes. A directed line also connects each critical section with its corresponding performance attributes. A dotted line is used to connect each performance attribute with a performance requirement which may be affected if the performance attribute is changed.

8.1.2 Algorithm to Identify All of the Mechanisms for the Propagation of Performance Changes Present in the Program

An important step of the maintenance technique to be described in Section 8.2 is the identification of all of the mechanisms for the propagation of performance changes present in the program. The algorithm described in this section identifies all of these mechanisms by invoking the algorithms for identifying each of these mechanisms described in Section 3.

Algorithm 8.1.2. Identification of the mechanisms for the propagation of performance changes present in the program.

Step 1. Apply Algorithm 3.1.1 to identify all sets of modules that are executable in parallel with a given module.

Step 2. Apply Algorithm 3.1.5 to identify the parallel execution sets for the program.

Step 3. Apply Algorithm 3.2.1 to identify the shared resource mechanism.

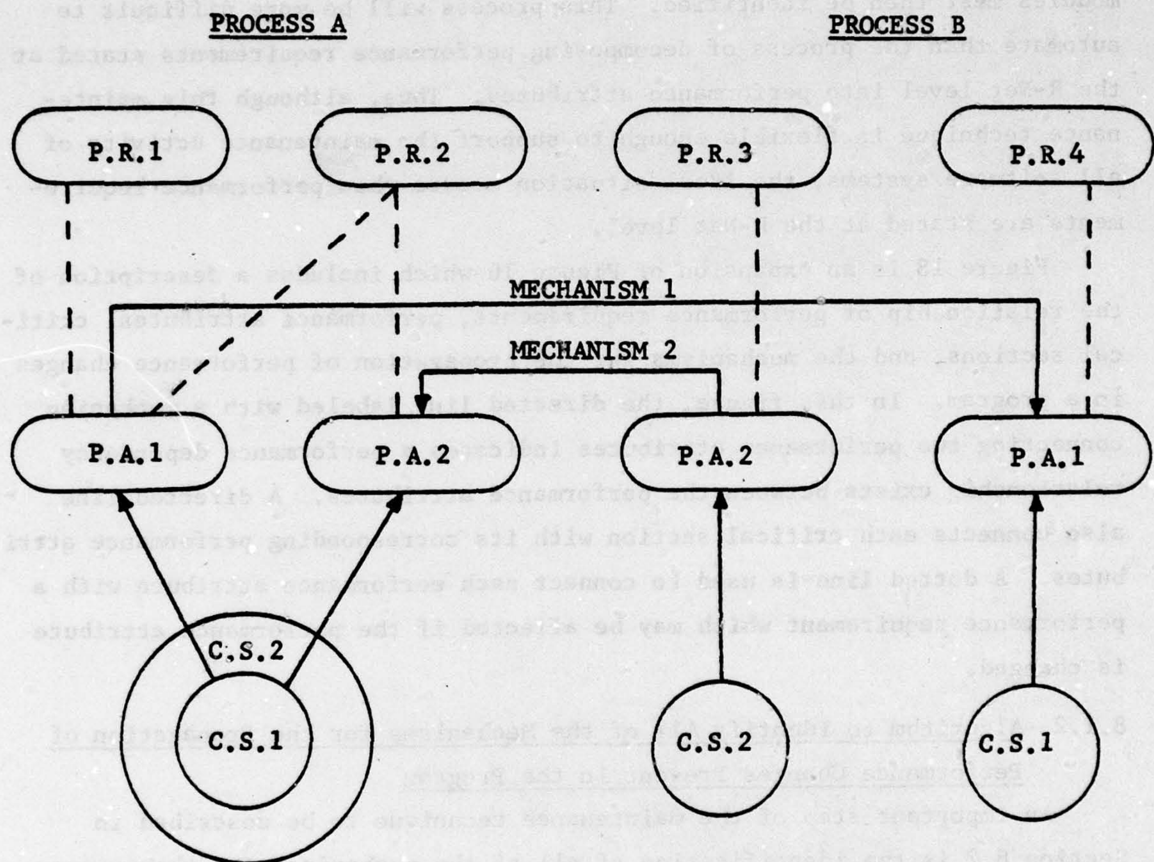


Figure 18. An example for illustrating a possible relationship among performance requirements (P.R.), performance attributes (P.A.), critical sections (C.S.) and the mechanisms for propagation of performance changes in a program.

Step 4. Apply Algorithm 3.3.1 to identify the interprocess communication mechanism.

Step 5. Apply Algorithm 3.4.1 to identify the called module mechanism.

Step 6. Apply Algorithm 3.5.1 to identify the shared data structures mechanism.

Step 7. Apply Algorithm 3.6.1 to identify the input-sensitivity mechanism.

Step 8. Apply Algorithm 3.7.1 to identify the execution priority sensitivity mechanism.

Step 9. Apply Algorithm 3.8.1 to identify the abstraction mechanism.

Theorem 8.1.2. Algorithm 8.1.2 identifies all of the mechanisms for the propagation of performance changes present in the program.

Proof: Algorithm 8.1.2 simply consists of invocations to Algorithms 3.1.1, 3.1.5, 3.2.1, 3.3.1, 3.4.1, 3.5.1, 3.6.1, 3.7.1, and 3.8.1. Now each of these algorithms identifies a particular mechanism for the propagation of performance changes as proven in Theorems 3.1.3, 3.1.7, 3.2.3, 3.3.3, 3.4.3, 3.5.2, 3.6.2, 3.7.3, and 3.8.3. Therefore, Algorithm 8.1.2 identifies all of the mechanisms for the propagation of performance changes present in the program.

8.1.3 Algorithm to Identify All of the Critical Sections in the Program

An important step of the maintenance technique to be described in Section 8.2 is the identification of all of the critical sections in the program. The algorithm described in this section identifies all of these critical sections by invoking the algorithms for identifying each critical section of a particular type described in Section 4.

Algorithm 8.1.3. Identification of all of the critical sections in the program.

Step 1. Apply Algorithm 4.1 to identify all critical sections of type one.

Step 2. Apply Algorithm 4.2 to identify all critical sections of type two.

Step 3. Apply Algorithm 4.3 to identify all critical sections of type three.

Step 4. Apply Algorithm 4.4 to identify all critical sections of type four.

Step 5. Apply Algorithm 4.5 to identify all critical sections of type five.

Step 6. Apply Algorithm 4.6 to identify all critical sections of type six.

Step 7. Apply Algorithm 4.7 to identify all critical sections of type seven.

Theorem 8.1.3. Algorithm 8.1.3 identifies all of the critical sections which are present in the program.

Proof: Algorithm 8.1.3 simply consists of invocations of Algorithms 4.1 to 4.7. Now each of the Algorithms 4.1 to 4.7 identifies a particular type of critical section present in the program as proven in Theorems 4.1 to 4.7. Therefore, Algorithm 8.1.3 identifies all of the critical sections present in the program.

8.1.4 Algorithm to Identify All of the Performance Attributes in the Program

An important step of the maintenance technique to be described in Section 8.2 is the identification of all of the performance attributes in the program described by the mechanisms for the propagation of performance changes.

Algorithm 8.1.4. Identification of all of the performance attributes in the program described by the mechanisms for the propagation of performance changes.

Step 1. For each module in the program and each critical section in the module's characterization, designate the corresponding performance attribute for the critical section to be a performance attribute for the module.

Step 2. For each module in the program, designate performance attribute 4 to be a performance attribute for the module.

Step 3. For each resource i and each module in T_i identified in Algorithm 3.2.1 designate performance attribute 7 for resource i to be a performance attribute for the module.

Step 4. For each data structure, designate performance attributes 8, 9, and 10 to be performance attributes for the data structure.

Step 5. For each module in the program which interfaces with the environment to handle input interrupts, designate performance attribute 11 to be a performance attribute for the module.

Theorem 8.1.4. Algorithm 8.1.4 identifies all of the performance attributes in the program described by the mechanisms for the propagation of performance changes.

Proof: The performance attributes were defined in terms of the mechanisms for the propagation of performance changes. Critical software sections were defined for performance attributes 1, 2, 3, 5, 6, 7 and 12. Step 1 of the algorithm simply designates the corresponding performance attributes for the critical sections to be performance attributes for the modules containing the critical sections. Now performance attributes 4, 8, 9, 10 and 11 do not possess critical sections. These performance attributes are identified by Steps 2, 4 and 5 of the algorithm by definition of the performance attributes in terms of the mechanisms for the propagation of performance changes. Finally, Step 3 of the algorithm identifies performance attribute 7 for those modules that do not directly reference the resource, but invoke a module which does utilize the resource by definition of performance attribute 7 in terms of the mechanisms for the propagation of performance changes.

Therefore, all of the performance attributes in the program described by the mechanisms for the propagation of performance changes are identified.

8.1.5 Algorithm to Identify All of the Virtual Performance Attributes in a Program

An important step of the maintenance technique to be described in Section 8.2 is the identification of all of the virtual performance attributes in the program.

Step 1. For each module involved in a PIR via mechanism 2 and each statement in the module which requests the resource in contention, designate virtual performance attribute of type one for the contended resource to be a virtual performance attribute for the module.

Step 2. For each module in the program and each statement in the module which invoked another module or an abstraction, designate virtual performance attribute of type two for the module or abstraction invoked to be a virtual performance attribute for the module invoking them.

Step 3. For each module in the program and each dependent iterative structure in the module, designate virtual performance attribute of type three for the dependent iterative structure to be a virtual performance attribute for the module.

Step 4. For each module in the program utilizing a data structure and each statement in the module which references the data structure, designate virtual performance attribute of type four for the data structure to be a virtual performance attribute for the module.

Step 5. For each dependent module involved in a PDR via mechanism 3 and each statement corresponding to a WAIT for the message, designate virtual performance attribute of type five for the message to be a virtual performance attribute for the module.

Theorem 8.1.5. Algorithm 8.1.5 identifies all of the virtual performance attributes in a program.

Proof: The proof of the theorem follows directly from the definition of firtual performance attributes of types one, two, three, four and five. Each of the steps of the algorithm identifies a virtual performance attribute according to the definition of the virtual performance attributes.

8.2 Formal Description of the Maintenance Technique

In this section, a maintenance technique for predicting which performance requirements are affected by program modifications will be formally described. The maintenance technique consists of two stages. The first stage consists of a lexical analysis of the program with respect to the proposed modification. In this stage, the program is analyzed with respect to the proposed modification, and the performance characterization of the program is compiled and saved in a data base. This characterization is then utilized in the second stage of the maintenance technique. The second stage consists of tracing the performance changes, i.e., the performance ripple effect which occurs as a consequence of the maintenance changes.

8.2.1 Lexical Analysis Stage of the Maintenance Technique

Step 1. Decompose the performance requirements for the program into the performance attributes which contribute to the preservation or violation of the performance requirements. Some guidelines for this decomposition are presented in Section 8.1.1.

Step 2. Apply Algorithm 8.1.2 to identify all of the mechanisms for the propagation of performance changes present in the program.

Step 3. Apply Algorithm 8.1.3 to identify all of the critical sections in the program.

Step 4. Apply Algorithm 8.1.4 to identify all of the performance attributes in the program.

Step 5. Apply Algorithm 8.1.5 to identify all of the virtual performance attributes in the program.

8.2.2 Algorithm to Trace the Ripple Effect of Performance Changes Among the Performance Attributes

The second stage of the maintenance technique to be described in Section 8.2.3 consists of tracing the performance changes which occur as a consequence of the program modification. An important step of this stage is the ability to trace the ripple effect of performance changes among the performance attributes. In this section, an algorithm will be described to accomplish this objective. The proof of this algorithm will be presented in Section 8.3 along with other major proofs of the algorithms composing the maintenance technique.

Algorithm 8.2.2. Identification of all of the performance attributes in a program which has been analyzed by the first stage of the maintenance technique described in Section 8.2.1 which may be affected as a consequence of modifying the performance attributes which are contained in set X.

Step 1. Select an element, x, in X which has not been selected before. If there are no new elements in X to select, then terminate.

Step 2. Utilizing the rules describing the performance dependency relationships among the performance attributes summarized in Section 6, identify all of the performance attributes in the program which may be affected by a modification of performance attribute x. Add these new performance attributes to X if they do not already appear.

Step 3. Utilizing the rules describing the performance dependency relationships between the performance attributes and the virtual performance attributes summarized in Section 6, identify all of the virtual performance attributes in the program which may be affected by a modification of performance attribute x. If x does not affect any virtual performance attributes, go to Step 5, else go to Step 4.

Step 4. For each of the virtual performance attributes identified for x in Step 3, apply one of algorithms 5.1 to 5.5 depending upon the type of virtual performance attribute identified, to identify all of the performance attributes affected by modification of the virtual performance attribute identified in Step 3. Add these new performance attributes to X if they do not already appear.

Step 5. Go to Step 1.

8.2.3 Tracing Stage of the Maintenance Technique

The second stage of the maintenance technique consists of tracing the performance changes, i.e., the performance ripple effect which occurs as a consequence of the maintenance changes. The input to the technique in this stage includes all of the information about the program collected and stored in a data base during the lexical analysis stage. The second stage of the maintenance technique identifies the performance requirements which are affected by the proposed program modification. This second stage consists of the following steps:

Step 1. Identify the critical sections which may be affected by the maintenance activity. The critical sections of the program were identified in Step 3 of Stage 1.

Step 2. For each of the critical sections identified in Step 1, determine the corresponding performance attributes which may be affected if the critical section is modified. The correspondence between critical sections and performance attributes was established in Section 4. Let X be the set of these performance attributes.

Step 3. Apply Algorithm 8.2.2 to identify all of the performance attributes which may be affected as a consequence of modifying the performance attributes contained in set X. Let X contain the set of all of these performance attributes.

Step 4. Identify those performance requirements which are affected by a modification of the performance attributes in set X. These performance requirements can be identified by the traceability of the decomposition of the performance requirements into the performance attributes in Step 1 of Stage 1.

The important steps of this maintenance technique are summarized and put into perspective within the maintenance process in Figure 19. From the example shown in Figure 18, the maintenance technique could be used to predict the performance implications of modifying CS.1 of Process A. In this example, PA.1 and PA.2 of Process A would be affected. Thus, performance requirements PR.1 and PR.2 would have to be retested to insure that they have not been violated. In addition, PA.1 of Process B would be affected via mechanism 1. Thus, PR.4 would also have to be retested to insure that it too has not been violated.

8.3 Proof of Algorithms Composing the Maintenance Technique

The maintenance technique described in Section 8.2 is very generalized and flexible in nature. Additional mechanisms, performance attributes, virtual performance attributes, critical sections and performance dependency relationship rules can easily be added to the maintenance technique to enable it to be applicable to any program. In this manner, the technique can be utilized to maintain any program. In this section, the remaining algorithms composing the maintenance technique will be proved to be correct.

8.3.1 Proof of the Algorithms Composing Stage One of the Maintenance Technique

In this section, the algorithms composing stage one of the maintenance technique will be proven correct.

Theorem 8.3.1. The first stage of the maintenance technique described in Section 8.2.1 identifies all of the mechanisms for the propagation of performance changes, critical sections, performance attributes and virtual performance attributes in the program.

Proof: Step 2 of Stage One of the maintenance technique invokes Algorithm 8.1.2 to identify the eight mechanisms for the propagation of performance changes defined in Section 3. Theorem 8.1.2 proves that all eight of these mechanisms are identified by Algorithm 8.1.2. Thus, the first stage of the maintenance technique identifies all of the mechanisms for the propagation of performance changes in the program.

Step 3 of Stage One of the maintenance technique invokes Algorithm 8.1.3 to identify all of the critical sections in the program. Theorem 8.1.3

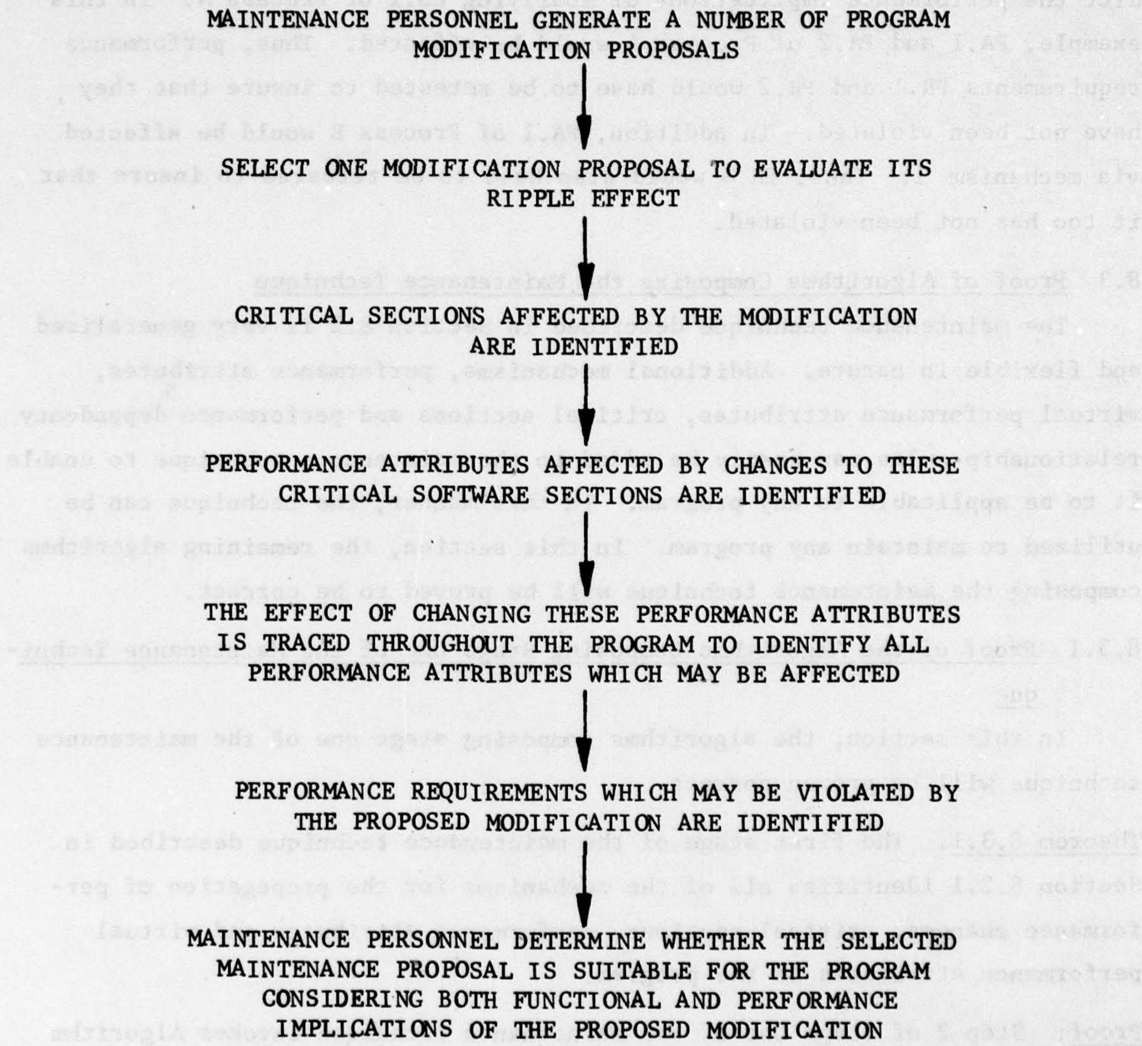


Figure 19. The framework of the performance ripple effect analysis technique for predicting which performance requirements are affected by the maintenance activity.

proves that all of the critical sections are identified by Algorithm 8.1.3. Thus, the first stage of the maintenance technique identifies all of the critical sections in the program.

Step 4 of Stage One of the maintenance technique invokes Algorithm 8.1.4 to identify all of the performance attributes in the program. Theorem 8.1.4 proves that all of the performance attributes are identified by Algorithm 8.1.4. Thus, the first stage of the maintenance technique identifies all of the performance attributes in the program.

Step 5 of Stage One of the maintenance technique invokes Algorithm 8.1.5 to identify all of the virtual performance attributes in the program. Theorem 8.1.5 proves that all of the virtual performance attributes are identified by Algorithm 8.1.5. Thus, the first stage of the maintenance technique identifies all of the virtual performance attributes in the program.

Therefore, the first stage of the maintenance technique identifies all of the mechanisms for the propagation of performance changes, critical sections, performance attributes and virtual performance attributes in the program.

8.3.2 Theorem for Algorithm 8.2.2

In this section, a proof of the correctness of Algorithm 8.2.2 will be presented. This algorithm is responsible for tracing the ripple effect of performance changes among the performance attributes.

Theorem 8.3.2. Algorithm 8.2.2 identifies all of the performance attributes in a program which has been analyzed by the first stage of the maintenance technique described in Section 8.2.1, which may be affected as a consequence of modifying the performance attributes which are contained in set X.

Proof: Let x be an arbitrary element of X. It must be shown that all performance attributes which may be affected as a consequence of modifying x are added to set X by the algorithm. Let y be an arbitrary performance attribute which may be affected as a consequence of modifying x. It must be shown that $y \in X$.

Now since the program has been analyzed by the first stage of the maintenance technique, Theorem 8.3.1 guarantees that all of the mechanisms for the propagation of performance changes, performance attributes and virtual performance attributes in the program have been identified. With this

information, all of the performance dependency relationships between performance attributes and virtual performance attributes can be determined by the defined rules which are summarized in Sections 5 and 6. The proof requires consideration of three cases.

Case 1. Assume there exists a rule summarized in Section 6.0 which describes a performance dependency relationship between performance attributes x and y. Then, Step 2 of the algorithm adds y to X.

Case 2. Assume there exists a performance dependency relationship between a virtual performance attribute z and performance attribute y and that there exists a rule summarized in Section 6 which describes a performance dependency relationship between performance attribute x and virtual performance attribute z. Then, Steps 3 and 4 of the algorithm add y to X.

Case 3. Assuming that neither case 1 nor case 2 is true, the fact that y may be affected as a consequence of modifying x implies there exists performance attributes $x, x_1, x_2, \ldots, x_k, y$, where x affects x_1, x_1 affects $x_2, \ldots, x_k$ affects y via either a rule summarized in Section 6 or a performance dependency relationship involving virtual performance attributes described in Section 5. In either case, Steps 1, 2, 3, 4 and 5 add $x_1, x_2, \ldots, x_k, y$ to set X.

Therefore, all performance attributes which may be affected as a consequence of modifying x are added to set X.

8.3.3 Proof of the Second Stage of the Maintenance Technique

The second stage of the maintenance technique described in Section 8.2.3 predicts which performance requirements are affected by a proposed program modification. This is accomplished by tracing the ripple effect of performance changes among the performance attributes. In this section a proof will be provided that this second stage of the maintenance technique accomplishes this objective.

Theorem 8.3.3. The second stage of the maintenance technique described in Section 8.2.3 identifies the performance requirements affected in a program based upon the performance characterization produced by the first stage of the maintenance technique which may be affected by a proposed modification.

Proof: Theorem 8.3.1 guarantees that all of the critical sections in the program were identified in the first stage of the maintenance technique. Step 1 of Stage Two of the maintenance technique then identifies all of these critical sections which may be affected by the maintenance activity.

Theorem 8.3.1 also guarantees that all of the performance attributes in the program were identified in the first stage of the maintenance technique. Step 2 of Stage Two of the maintenance technique then identifies all of the performance attributes which correspond to the critical sections identifed in Step 1 of Stage Two of the maintenance technique. This correspondence between performance attributes and critical sections was defined in Section 4.

Step 3 of Stage Two of the maintenance technique then invokes Algorithm 8.2.2 to identify all of the performance attributes in the program which may be affected as a consequence of modifying the performance attributes identified in Step 2 of Stage Two of the maintenance technique. Theorem 8.3.2 guarantees that all of these performance attributes will be identified.

Step 4 of Stage Two of the maintenance technique then identifies all of the performance requirements which are affected by a modification of the performance attributes identified in Step 3 of Stage Two of the maintenance technique. These performance requirements can be identified directly by the traceability of the decomposition of the performance requirements into the performance attributes accomplished in Step 1 of Stage One of the maintenance technique.

Therefore, the second stage of the maintenance technique identifies the performance requirements which may be affected by a proposed modification.

8.4 Application of the Maintenance Technique to the Retesting Phase of the Maintenance Process

After the maintenance changes have been completely implemented, this technique can provide a significant contribution to the application of retesting the program to verify that the performance requirements for the program have not been violated by the maintenance effort. The retesting of large-scale complex programs requires a great deal of time, effort and expense. Thus, any savings resulting from this maintenance technique will clearly justify its use.

During the early stages of the maintenance process, this technique was utilized as an aid in developing criteria for maintenance personnel to evaluate alternate program modifications from a performance perspective. Basically, this involved the worst-case identifications of performance requirements which might be affected by the program modifications. After a program modification has been selected and completely implemented, the maintenance technique can substantially refine its analysis and determine more accurately which performance requirements may have been affected by the program modifications. This is accomplished by determining whether or not a performance attribute is actually affected before implicating other performance attributes involved in a performance dependency relationship with the given attribute. In other words, if a dependency relationship exists between performance attributes 1 and 2, performance attribute 2 need not be examined for changes if it has been determined that performance attribute 1 is not affected by the maintenance activity. Thus, the preliminary results of some of the early retesting efforts may be decisive in the determination of the scale of retesting which remains to be done. This technique is summarized and put into perspective within the maintenance process in Figure 20. It should be noted that if a violation of a performance requirement occurs, it requires further software maintenance in order to satisfy the performance requirement, and the entire process must be repeated.

9.0 FIGURE OF MERIT FOR THE COMPLEXITY OF A PROGRAM MODIFICATION

The previous sections of this report contained the development of a maintenance technique for predicting which performance requirements in the program may be affected by a proposed software modification. They also illustrated how the maintenance technique can help retest the program to determine whether its performance requirements have been violated by the maintenance effort after the maintenance changes have been implemented.

Another very significant product of the analysis of ripple effect is the computation of a figure-of-merit for the complexity of a proposed program modification. One such figure-of-merit is defined in [8] as follows:

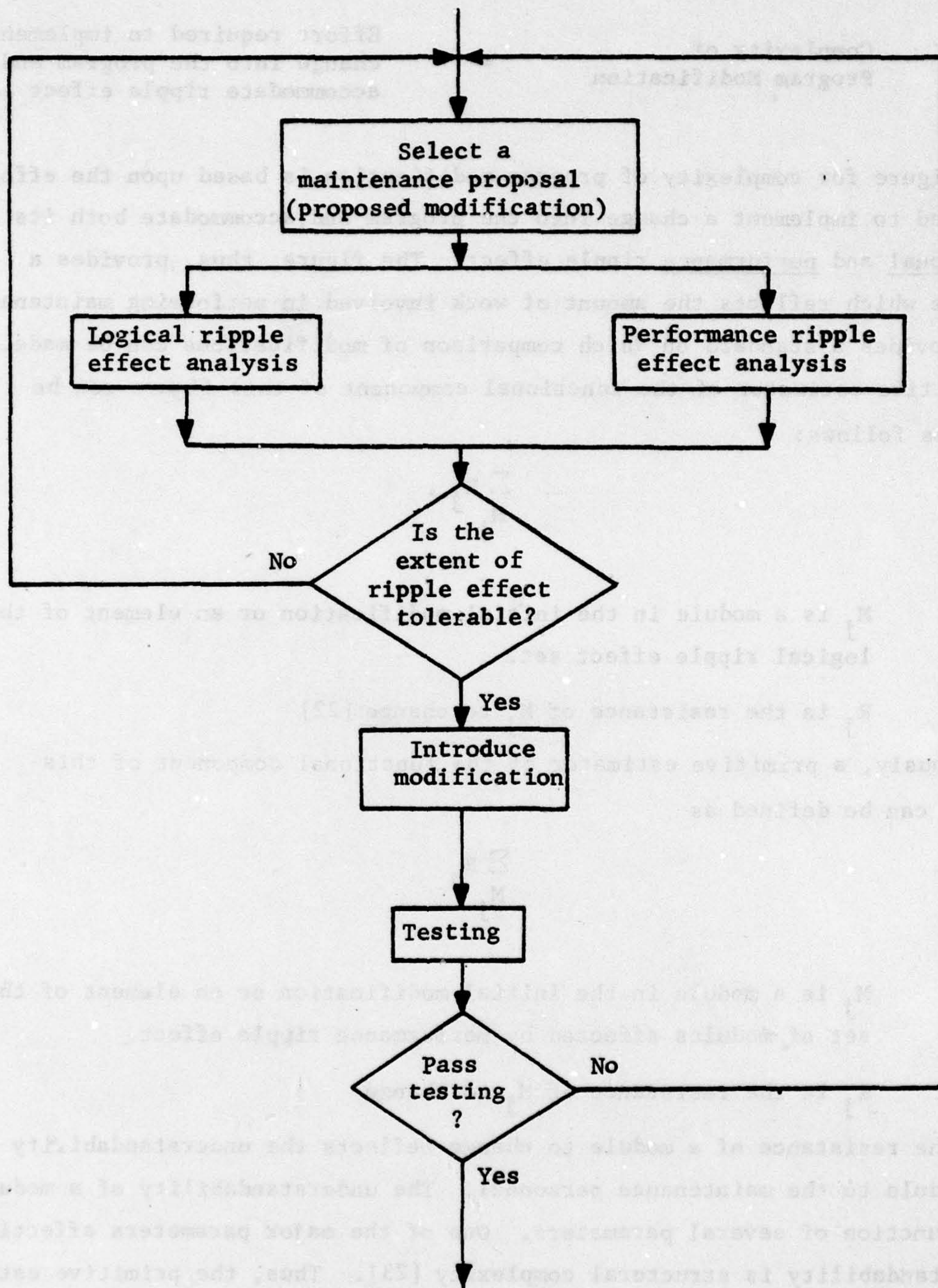


Figure 20. The performance ripple effect analysis technique as a part of the maintenance process.

$$\text{Complexity of Program Modification} = \text{Effort required to implement change into the program and accommodate ripple effect}$$

This figure for complexity of program modification is based upon the effort required to implement a change into the program and accommodate both its functional and performance ripple effect. The figure, thus, provides a measure which reflects the amount of work involved in performing maintenance and provides a standard on which comparison of modifications can be made. A primitive estimator of the functional component of this figure can be given as follows:

$$\sum_{M_j} R_j ,$$

where

M_j is a module in the initial modification or an element of the logical ripple effect set.

R_j is the resistance of M_j to change [22]

Analogously, a primitive estimator of the functional component of this figure can be defined as

$$\sum_{M_j} R_j$$

where

M_j is a module in the initial modification or an element of the set of modules affected by performance ripple effect.

R_j is the resistance of M_j to change

The resistance of a module to change reflects the understandability of the module to the maintenance personnel. The understandability of a module is a function of several parameters. One of the major parameters affecting understandability is structural complexity [23]. Thus, the primitive estimator presented here can be based upon the structural complexity of the modules involved in the maintenance effort.

10.0 FUTURE RESEARCH AND CONCLUSION

10.1 Dynamic Analysis

The significance of this performance ripple effect analysis technique is its ability to trace repercussions introduced by maintenance changes and predict which performance requirements may be affected by the program modifications. This information is very valuable to maintenance personnel but its significance can be even greater if it is supplemented by a profile of the dynamic behavior of the program. This profile can provide maintenance personnel with performance information about the program enabling them to identify performance requirements which are close to being violated. This information coupled with that predicting which performance requirements may be affected by a program modification provides maintenance personnel with strong guidelines for selecting among alternative program modifications.

The profile of the dynamic behavior of a program also plays a significant role in the retesting portion of the maintenance phase to insure that the program modifications have not resulted in violation of any performance requirements. After the maintenance modifications have been implemented and the resultant changes in performance analyzed, this information can be used along with the profile of the dynamic behavior of the program to determine the scale of retesting which remains to be done. For example, if a process has a performance requirement stating that it completes execution in 100 units, it will not be affected by a performance change of about 5 units if the program's profile indicates it is currently completing execution in 75 units.

The profile of the dynamic behavior of the program is, thus, important in the maintenance phase. It should be noted that the profile itself is dynamic since it only reflects the dynamic behavior in the current environment. Nevertheless, it is important in performance investigations since the performance of the program is also dependent of the current environment. It is, thus, meaningless to analyze performance considerations utilizing a profile based upon a different operating environment.

More research is needed in the identification of appropriate dynamic measurements to be included in this profile. It is seen from previous sections that measurements pertaining to resource utilization, execution times

of critical software sections, system overhead, and degree of saturation of data structures provide the most meaningful information for the maintenance process. The feasibility of collecting many of these measurements in large-scale software systems has already been demonstrated. For example, JAVS provides a facility for capturing the execution time spent in individual modules [24]. The Program Evaluator and Test System developed by Stucki [25] also provides relative timing on the subroutine level. System overhead has also been studied for some time, and both hardware and software measuring techniques exist to identify many of its sources. Measurements pertaining to resource utilization, such as data structures, have also been recorded using software probes [26]. Data pertaining to the time when a resource is requested and when that request is actually satisfied has been collected on large-scale software systems with a degradation of system performance due to resources committed to probe operation not exceeding 5%. For example, Figure 21 illustrates the types of measurements that can be gathered for an executing process. In the figure, execution times between requests as well as probabilities that particular branches from decision nodes are executed appear in the graph [27]. These types of measurements would be important in formulating the profile of the dynamic behavior of the program most applicable to the maintenance process. More research is needed in the identification of other appropriate dynamic measurements to be included in this profile.

10.2 Refined Estimator of the Complexity of Program Modification from a Performance Perspective

In section 9 a primitive estimator of the performance component of a figure for the complexity of a proposed program modification was described. This estimator was based upon the performance ripple effect of the proposed modification and the structural complexity of the modules involved in the maintenance effort. This figure can be enhanced by the addition of other factors which contribute to the complexity of program modification from a performance perspective.

One such factor involves the profile of the dynamic behavior of the program. The performance of a program is a subject which ranges from quantitative analyses to qualitative judgements [28]. Thus, the current behavior of the program in perspective to its performance requirements can

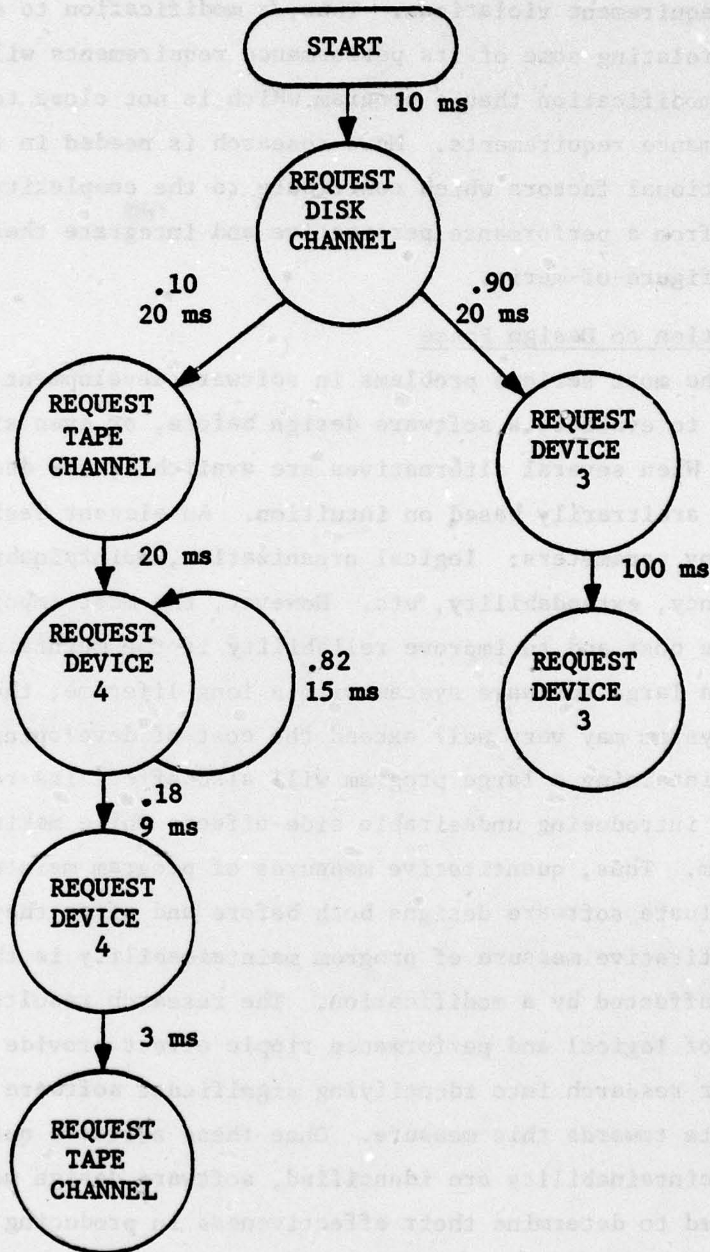


Figure 21. A directed graph representing resource utilization of a process.

provide insight into the degree of performance changes that can be tolerated without a performance requirement violation. For example, if the program is operating near a point of saturation, any performance changes could lead to performance requirement violations. Thus, a modification to a program which is close to violating some of its performance requirements will require a more complex modification than a program which is not close to violating any of its performance requirements. More research is needed in this area to identify additional factors which contribute to the complexity of program modification from a performance perspective and integrate these factors into a meaningful figure-of-merit.

10.3 Application to Design Phase

One of the most serious problems in software development is the lack of methodologies to evaluate a software design before, or even after, it is implemented. When several alternatives are available, the design decision is made quite arbitrarily based on intuition. An elegant design of a system depends on many parameters: logical organization, maintainability, compactness, efficiency, extendability, etc. However, the most important consideration to reduce cost and to improve reliability is the maintainability of the system. For a large software system with a long lifetime, the cost of maintaining the system may very well exceed the cost of developing the system for delivery. Maintaining a large program will also affect its reliability by unconsciously introducing undesirable side-effects while making a modification in the program. Thus, quantitative measures of program maintainability are needed to evaluate software designs both before and after they are implemented.

One quantitative measure of program maintainability is the extent that a program is affected by a modification. The research results obtained in the analysis of logical and performance ripple effect provide an excellent foundation for research into identifying significant software quality factors that contribute towards this measure. Once these software quality factors that affect maintainability are identified, software design methodologies can be analyzed to determine their effectiveness in producing maintainable software. The results of this investigation may lead to the development of a new software design methodology which produces programs with good maintenance characteristics.

10.4 Conclusion

The process of developing complex large scale software systems possessing performance requirements is costly, excessively time-consuming, and difficult to manage. This process frequently leads to systems which are unreliable, not responsive to user-requirements, and logically too obscure to be readily analyzed or maintained. Yet these software systems must be maintained, and the magnitude of this maintenance in terms of the total software effort is very large. Some projections indicate that the magnitude of this maintenance cost may ultimately result in a five or ten to one ratio of maintenance costs to research and development costs when viewed over the total life cycle of a typical system [29]. Thus, maintenance techniques are needed that are specifically designed to predict the effect of software modifications and indicate test cases required for program retesting. The maintenance technique presented in this report is designed with these objectives in mind and should significantly aid maintenance personnel in maintaining software systems.

11.0 REFERENCES

[1] Boehm, B. W., "Software Engineering: R&D Trends and Defense Needs," in Research Directions in Software Technology, Wegner, P. (ed.), MIT Press, 1979, pp. 44-86.

[2] Boehm, B. W., "Software and Its Impact: A Quantative Assessment," Datamation, May 1973, pp. 48-59.

[3] Myers, W., "The Need for Software Engineering," Computer, Volume 11, No. 2, February 1978, pp. 12-26.

[4] Munson, J. B., "Software Maintainability: A Practical Concern for Life-Cycle Costs," Proc. COMPSAC 78, November 1978, pp. 54-60.

[5] Zelkowitz, M. V., "Perspectives on Software Engineering," ACM Computing Surveys, Vol. 10, No. 2, June 1978, pp. 197-216.

[6] Lientz, B. P. and Swanson, E. B., "Characteristics of Application Software Maintenance," Comm. ACM, Vol. 21, No. 6, June 1978, pp. 466-471.

[7] Elshoff, J. L., "An Analysis of Some Commercial PL/1 Programs," IEEE Trans. on Software Engineering, Vol. SE-2, No. 2, June 1976, pp. 113-120.

[8] Yau, S. S., Collofello, J. S., and MacGregor, T., "Ripple Effect Analysis of Software Maintenance," Proc. COMPSAC 78, Nov. 1978, pp. 60-65.

[9] Rye, P., Bamberger, F., Ostanek, W., Brodeur, N. and Goode, J., Software Systems Development: A CSDL Project History, RADC-TR-77-213, pp. 33-41, A042186.

[10] Goodenough, J. B., and Zara, R. V., "The Effect of Software Structure on Software Reliability, Modifiability, and Reusability: A Case Study and Analysis," Softech Incorporated, July 1974, p. 82.

[11] McCall, J. A., Richards, P. K., and Walters, G. F., Factors in Software Quality, Volumes I, II, and III, General Electric Company, June 1977, pp. 2-3, 3-5, 7-9.

[12] Coullon, H., Isle, R., and Lohr, K., "Dynamic Restructuring in an Experimental Operating System," Proc. 3rd Int'l Conf. on Software Engineering, 1978, pp. 295-304.

[13] Ringland, G. and Trice, A. R., "Pilot Implementations of Reliable Systems," Software Practice and Experience, Vol. 8, May-June 1978, pp. 323-339.

[14] Yourdon, E. and Constantine, L., Structured Design, Yourdon Inc., 1976, p. 392.

[15] Haney, F. M., "Module Connection Analysis--A Tool for Scheduling Software Debugging Activities," Proc. Fall Joint Computer Conf., 1972, pp. 173-179.

[16] Boyd, D. L., and Pizzarello, A., "Introduction to the Wellmade Design Methodology," Proc. 3rd Int'l Conf. on Software Engineering, 1978, pp. 94-100.

[17] Yau, S. S. and Collofello, J. S., Performance Considerations in the Maintenance Phase of Large-Scale Software Systems, RADC-TR-79-129, June, 1979, 44 pages, A072380.

[18] Yau, S. S., Collofello, J. S., and Hsieh, C. C., Ripple Effect Analysis for Large-Scale Software Maintenance--A Handbook, RADC Technical Report, to be published.

[19] Swanson, E. B., "The Dimensions of Maintenance," Proc. 2nd Int'l Conf. on Software Engineering, October 1976, pp. 492-497.

[20] Belford, P. C., Donahoe, J. D., and Heard, W. J., "An Evaluation of the Effectiveness of Software Engineering Techniques," Digest of Papers, COMPCON 77 (Fall), pp. 259-269.

[21] Alford, M. W. and Burns, I. F., "R-Nets: A Graph Model for Real-Time Software Requirements," Proc. MRI Symp. on Software Engineering, Polytechnic Institute of New York, 1976, pp. 97-107.

[22] Yau, S. S. and Collofello, J. S., "Some Stability Measures for Software Maintenance," Proc. COMPSAC 79, November 1979, pp. 674-679.

[23] McCabe, T. J., "A Complexity Measure," IEEE Trans. on Software Engineering, Vol. SE-2, No. 4, December 1976, pp. 308-320.

[24] JAVS Technical Report Reference Manual, RADC-TR-77-126, Vol. 2, April 1977, Vol I A040103, Vol II A040104, Vol III A041048.

[25] Stucki, L. G., "Automatic Generation of Self-Metric Software," Proc. Symposium on Computer Software Reliability, 1973, pp. 94-101.

[26] Yau, S. S. and Ramey, J. L., Dynamic Monitoring for Linear List Data Structures, RADC-TR-79-128, June 1979, 111 pages, A072381.

[27] Anderson, J. W. and Browne, J. C., "Graph Models of Computer Systems: Application to Performance Evaluation of an Operating System," Proc. Int'l Symp. on Computer Performance Modeling, Measurement, and Evaluation, 1976, pp. 166-178.

[28] Kuck, D. J. and Kumar, B., "A System Model for Computer Performance Evaluation," Proc. Int'l Symp. on Computer Performance Modeling, Measurement, and Evaluation, 1976, pp. 187-199.

[29] DeRoze, B. C. and Nyman, T. H., "The Software Life Cycle--A Management and Technological Challenge in the Department of Defense," IEEE Trans. on Software Engineering, Vol. SE-4, No. 4, July 1978, pp. 309-326.

MISSION
of
Rome Air Development Center

RADC plans and executes research, development, test and selected acquisition programs in support of Command, Control Communications and Intelligence (C^3I) activities. Technical and engineering support within areas of technical competence is provided to ESD Program Offices (POs) and other ESD elements. The principal technical mission areas are communications, electromagnetic guidance and control, surveillance of ground and aerospace objects, intelligence data collection and handling, information system technology, ionospheric propagation, solid state sciences, microwave physics and electronic reliability, maintainability and compatibility.

DAT

FILM

6-