

MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS 1963-A

12



RADC-TR-83-262
Final Technical Report
February 1984

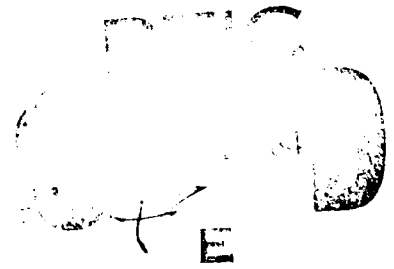
AD-A143 763

METHODOLOGY FOR SOFTWARE MAINTENANCE

Northwestern University

Stephen S. Yau

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED



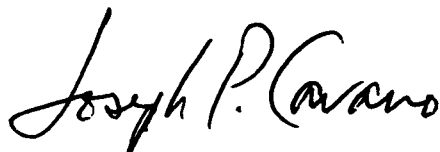
MM FILE COPY

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

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-83-262 has been reviewed and is approved for publication,

APPROVED:



JOSEPH P. CAVANO
Project Engineer

APPROVED:



RONALD S. RAPOSO
Acting Technical Director
Command and Control Division

FOR THE COMMANDER:



JOHN A. RTIZ
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 (COEE) Griffiss AFB NY 13441. This will assist us in maintaining a current mailing list.

Do not return copies of this report unless contractual obligations or notices on a specific document requires that it be returned.

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER RADC-TR-83-262	2. GOVT ACCESSION NO. A143763	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) METHODOLOGY FOR SOFTWARE MAINTENANCE	- 23	5. TYPE OF REPORT & PERIOD COVERED Final Technical Report Apr 80 - Sep 82
		6. PERFORMING ORG. REPORT NUMBER N/A
7. AUTHOR(s) Stephen S. Yau		8. CONTRACT OR GRANT NUMBER(s) F30602-80-C-0139
9. PERFORMING ORGANIZATION NAME AND ADDRESS Northwestern University Evanston IL 60201		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS 62702F 55812018
11. CONTROLLING OFFICE NAME AND ADDRESS Rome Air Development Center (COEE) Griffiss AFB NY 13441		12. REPORT DATE February 1984
		13. NUMBER OF PAGES 312
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: Joseph P. Cavano (COEE)		
19. KEY WORDS (Continue on reverse side if necessary and identify by block number) software modification software testing software maintenance software measurement software metrics logical ripple effect analysis		
20. ABSTRACT (Continue on reverse side if necessary and identify by block number) Improved techniques for specifying and implementing software modifications were developed including logical ripple effect analysis, logical and performance stability measures, and effective testing for software maintenance. An experiment was performed to analyze logical stability measurements.		

ABSTRACT

This report documents the research performed under RADC Contract No. F30602-80-C-0139 by Northwestern University for developing effective methodologies for software maintenance. This contract is a follow-on to Contract No. F30602-76-C-0397 and focuses on refining, expanding and automating software maintenance concepts and techniques developed under the previous contract.

During this contract period, significant progress was made in developing techniques for specifying and realizing software modification proposals, logical ripple effect analysis and module revalidation after modification. These techniques and the performance ripple effect analysis technique developed during the previous contract period were demonstrated using a DEC VAX 11/780 computer. In addition, a number of software metrics related to modifiability, such as measures for logical and performance stability, module strength and coupling, were developed. Limited experiments for validating the logical stability measures were performed.

In this report, research results which were presented in published papers are summarized, and unfinished and unpublished work is presented in detail. Publications, and technical personnel related to this project are also summarized. Published papers presenting the work supported by this contract are included in the Appendix.

Key Words - Software maintenance, maintenance methodology, specification and realization of software modification proposals, logical and performance ripple effect analysis, program revalidation, techniques, software metrics, stability, module strength and coupling, validation experiments.



Accession For	
NTIS GRA&I	<input checked="" type="checkbox"/>
DTIC TAB	<input type="checkbox"/>
Unannounced	<input type="checkbox"/>
Justification	
By _____	
Distribution/ _____	
Availability Codes	
Dist	Avail and/or Special
A-1	

TABLE OF CONTENTS

	ABSTRACT	i
	LIST OF ILLUSTRATIONS	v
	LIST OF TABLES	vii
1.0	INTRODUCTION	1
2.0	SOFTWARE MAINTENANCE PROCESS AND ASSOCIATED QUALITY FACTORS	3
2.1	The Activities Of Software Maintenance	3
2.2	Tools And Techniques For A Software Maintenance Environment	8
2.3	Quality Factors Affecting Software Maintenance	11
3.0	SPECIFICATION OF SOFTWARE MAINTENANCE PROPOSALS	13
3.1	A Model Of Software Systems For Software Maintenance	18
3.1.1	Background	18
3.1.2	Graph Rewriting Systems	20
3.1.2.1	Graph Rewriting	20
3.1.2.2	Definition Of A Labelled Graph	24
3.1.3	The Intraphase Model	24
3.1.3.1	Software Components	25
3.1.3.2	Component Interfaces	36
3.1.4	The Interphase Model	40
3.2	Construction Of The Software Model	45
3.2.1	Construction Of The Intraphase Model	45
3.2.1.1	Definition Procedure	47
3.2.1.2	An Example For Constructing An Intraphase Model (An RSL Subset)	48
3.2.1.3	Implementation Of The Intraphase Model	51
3.2.2	Construction Of The Interphase Model	52
3.2.2.1	Definition Of The Interphase Model	55
3.2.2.2	Implementation Of The Interphase Model	56
3.3	A Technique For Specifying Software Modification Proposals	57
3.3.1	Intraphase Tracing	58
3.3.1.1	An Example Of RSL Modification	61
3.3.1.2	Assertions To Control Intraphase Tracing	63
3.3.2	Interphase Tracing	65
3.3.2.1	An Example Of Interphase Tracing	65
3.4	Discussion And Future Work	71
4.0	REALIZATION OF SOFTWARE MAINTENANCE PROPOSALS	75
4.1	Overview	75
4.2	The Program Representation	79
4.2.1	Data Flow Extensions To The Basic Representation	81

4.2.2	The Construction Of The Representation	84
4.3	The Program Slicer	85
4.3.1	The Concept Of Program Slicing	85
4.3.2	Algorithms For Syntax-Directed Program Slicing	89
4.3.3	Enhancements	95
4.4	The Syntax-directed Editor	96
4.4.1	Incremental Editing	98
4.4.2	Legitimate Operations	99
4.4.3	Incremental Analysis	100
4.4.4	Incremental Update Of Data Flow Information	104
4.4.5	Interactive Pretty-printing	106
4.5	Software Development	106
4.6	Discussion And Future Work	110
5.0	RIPPLE EFFECT ANALYSIS	113
5.1	Logical Ripple Effect Analysis Technique	113
5.1.1	Intramodule Error Flow Model	118
5.1.1.1	Block Error Characteristics	120
5.1.1.2	Construction Of Intramodule Error Flow Model	124
5.1.1.3	Intramodule Error Flow Tracing	125
5.1.2	Intermodule Error Flow Model	132
5.1.2.1	Module Error Characteristics	134
5.1.2.2	Generation Of Module Error Characteristics	138
5.1.2.3	Update Block Error Characteristics	140
5.1.3	Logical Ripple Effect Identification	143
5.1.3.1	Error Flow Tracing	144
5.1.3.2	Intermodule Error Flow Tracing	145
5.1.3.3	Error Flow Tracing Algorithm	146
5.1.3.4	Logical Ripple Effect Derivation	150
5.1.4	Logical Ripple Effect Analysis Technique	153
5.1.5	Experiments	156
5.1.6	Discussion And Future Work	159
5.2	The Performance Ripple Effect Analysis Technique	160
5.2.1	Experimentation	161
5.2.2	Discussion	163
6.0	EFFECTIVE TESTING FOR SOFTWARE MAINTENANCE	165
6.1	The Module Revalidation Technique	166
6.1.1	Derivation Of The Input Partition	168
6.2	Reusability Of Original Test Cases	172
6.3	Assignment Of Original Test Set To The Input Partition Classes	173
6.4	Selection Of Original Test Cases For Execution	174
6.4.1	Necessary Information For Test Selection	175
6.4.2	Overview Of Selective Test Execution	178
6.4.3	Algorithm To Select Test Cases	180
6.4.4	An Example	181
6.5	Test Case Generation And Execution	185
6.6	Output Validation Phase	186
6.7	Debugging	187
6.8	Discussion And Future Work	189

7.0	METRICS RELATED TO SOFTWARE MAINTENANCE	192
7.1	Logical Stability Measure	193
7.1.1	Logical stability measure for modules	193
7.1.2	Logical stability measure for programs	195
7.2	Performance Stability Measure	196
7.3	Design Stability Measure	198
7.4	Module Strength and Coupling	200
7.4.1	Estimating Data Object Interaction	201
7.4.2	Definition of Intra-Module Strength Metric	204
7.4.3	Definition of Inter-Module Coupling Metric	205
7.5	Validation of the Logical Stability Measure	209
7.5.1	Experimental Procedures	209
7.5.1.1	Program Selection	210
7.5.1.2	Modification Proposal Generation	210
7.5.1.3	Quantification of the Realized Modifications	211
7.5.1.4	Actual Ripple Effect Estimation and Normalization	211
7.5.1.5	Statistical Methods Used in Analysis of the Results	213
7.5.2	Analysis of the Experimental Results	214
7.5.3	Discussion	215
7.6	A Unified and Efficient Approach for Logical Ripple Effect Analysis Used in Metrics Calculation	225
7.6.1	Formalization of Logical Ripple Effect Analysis	227
7.6.2	Logical Ripple Effect Analysis for Metrics Calculation	230
7.6.2.1	No Control Flow - No Sharing	231
7.6.2.2	No Control Flow - Sharing	233
7.6.2.3	Control Flow - Tracing	236
7.6.3	Conclusion	237
7.7	Discussion and Future Work	238
8.0	REFERENCES	239
9.0	PUBLICATIONS AND PRESENTATIONS	246
9.1	Papers	246
9.2	Presentations	247
9.3	Technical Reports	248
9.4	Dissertation And Theses	248
10.0	TECHNICAL PERSONNEL	250
11.0	APPENDIX	251

LIST OF ILLUSTRATIONS

Figure 2.1.	The software development process	4
Figure 2.2.	The software maintenance methodology	8
Figure 3.1.	The users' view of a software system	13
Figure 3.2.	The programmers' view of the same software system	15
Figure 3.3.	Equivalence preserving requirements for reliable software modification	16
Figure 3.4.	An example for rewriting a string	22
Figure 3.5.	An example for rewriting a graph.	23
Figure 3.6.	An example of the control flow representation	28
Figure 3.7.	Notational abbreviations for the standard graph patterns	29
Figure 3.8.	Abbreviated representation of the previous example shown in Figure 3.6	30
Figure 3.9.	An example of data flow representation.	32
Figure 3.10.	An example of data structure representation .	34
Figure 3.11.	An example of a component interface graph with no PARAMETERS subcomponent	40
Figure 3.12.	A process structure using Jackson's design methodology	42
Figure 3.13.	The model representation of the JDM design shown in Figure 3.12.	43
Figure 3.14.	Part of the interphase model between requirements and design	44
Figure 3.15.	RSL R_Net and associated alphas	62
Figure 3.16.	The MODEL representation of control flow of the RSL example	63
Figure 3.17.	Consequences of possible combinations of good and bad code with good and bad assertions . .	64
Figure 3.18.	The abstract graph representing a software system	66
Figure 3.19.	Tracing rules between two phases of an abstract software system	66
Figure 3.20.	The next phase of the abstract software system.	67
Figure 3.21.	A new right-hand side for rule R4	69
Figure 3.22.	An alternative right-hand side for rule R4 .	71
Figure 4.1.	The procedure for incremental program modification	76
Figure 4.2.	The structure of the system for incremental program modification	78
Figure 4.3.	(a) Portions of the program to be modified, (b) portions of the slice constructed for the variable COUNT	88

Figure 4.4.	A part of a legitimate operation table . . .	101
Figure 4.5.	Insertion of a local variable	103
Figure 4.6.	An example to show a sequence of cursor movements	107
Figure 4.7.	The communication pattern of the integrated tools	109
Figure 5.1.	An example program	126
Figure 5.2.	The error characteristics of the blocks in roots in the example program shown in Figure 5.1.	127
Figure 5.3.	Intramodule error flow tracing in roots in the example program shown in Figure 5.1 . . .	131
Figure 5.4.	The module error characteristics of roots in the example program shown in Figure 5.1. .	140
Figure 5.5.	Error flow tracing in the example program shown in Figure 5.1	149
Figure 6.1.	An overview of module revalidation	167
Figure 6.2.	The cause/effect graph specification of an example program	170
Figure 6.3.	The source code of the example program with the specification shown in Figure 6.2	171
Figure 6.4.	Original test cases prepared for the program in Figure 6.3	174
Figure 6.5.	The program graph with reaching set information for the program in Figure 6.3 . .	182
Figure 6.6.	Result of test selection on the program in Figure 6.3 with test cases in Figure 6.4: (a) symbolic execution tree, and (b) contents of test information table	183
Figure 6.7.	Decision table containing partition classes derived from the specification in Figure 6.2 and the program in Figure 6.3	188

LIST OF TABLES

Table 7.1.	Logical stability measures for each module of the target programs used in the experiment	216
Table 7.2.	Correlation analysis on logical stability for individual modules	222
Table 7.3.	The summary correlation analysis of logical stability for all modules in the experiment	224

1.0 INTRODUCTION

This report summarizes the research performed under Contract No. F30602-80-C-0139 by Northwestern University for Rome Air Development Center during the period from April 23, 1980 to November 30, 1982.

The original objective of this effort was to conduct exploratory development of techniques for the design, implementation, validation and evaluation of reliable and maintainable software systems. This effort was intended to be a follow-on to Contract No. F30602-76-C-0397, "Self-Metric Software" [YAUB0a, 80b, 80c], and would focus on refining, expanding and automating software maintenance concepts and techniques developed under the previous contract.

The original effort was planned for a period of three years, starting April 23, 1980. However, because of some difficulty in continued funding, this project was re-scoped in September, 1981 and had a lower level of funding starting FY82. This project starting October, 1981 was re-directed as follows: to complete the development of those techniques which could be completed in FY82, and to complete the development and perform some preliminary validation of the logical stability measures of programs for measuring the resistance of the programs to logical ripple effect due to modifications. In this report,

research results which have been presented in previous papers and interim technical reports are summarized, and unfinished and unpublished work is presented in more detail. Publications, presentations and technical personnel related to this project are also summarized.

During this contract period, we have made significant progress in developing techniques for specifying and realizing software modification proposals, logical ripple effect analysis and module revalidation after modification. These techniques and the performance ripple effect analysis technique developed during the last contract period have been demonstrated using a DEC VAX 11/780 computer. In addition, we have developed a number of software metrics related to modifiability, such as measures for logical and performance stability, module strength and coupling. Limited experiments for validating the logical stability measures have also been performed.

2.0 SOFTWARE MAINTENANCE PROCESS AND ASSOCIATED QUALITY FACTORS

The software maintenance phase is the most time-consuming and costly part of the software life cycle [BOEH73], [ZELK78], [LIEN80]. However, the activities carried out during this phase are deeply affected by the process of software development, since the purpose of software maintenance is to modify the products of the software development process.

2.1 The Activities Of Software Maintenance

We conceive the software development process as shown in Figure 2.1. The first activity of software development is to study the application area and define the requirements for a new software system for the particular application problem. This activity involves the participation of representatives from both the users of the software system and from the software development organization. The second activity (or set of activities) is known as software design. This activity may involve the definition of several intermediate stages during which a system is being developed to meet its requirements. These intermediate stages may be known as, for example, architectural design, subsystem design and module design. This activity is normally carried out exclusively by members of the software development organization, although some user

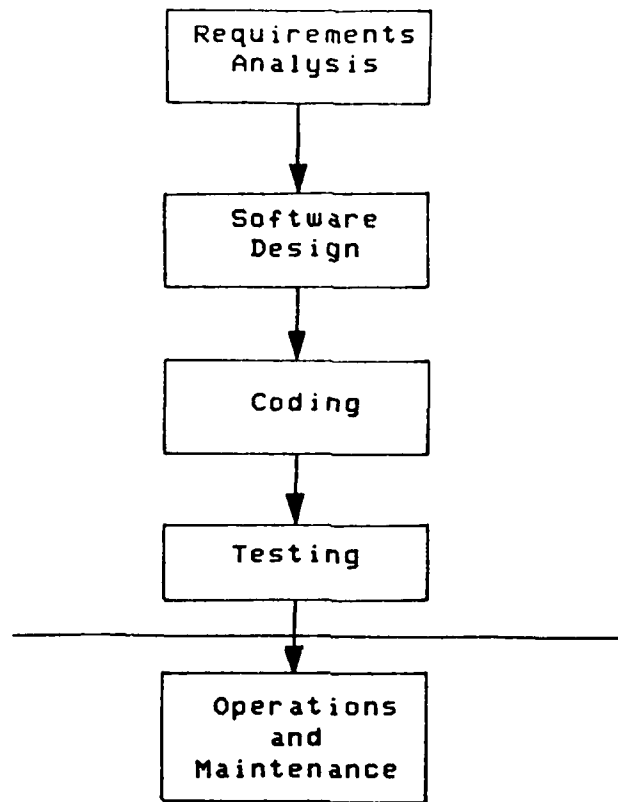


Figure 2.1. The software development process.

organizations may continue their involvement through various stages of design. The next activity of software development is to organize the software system design into one or more related programs, together with associated data files. This activity is known as the programming or coding phase. The final activity of software development is to test the implementation of the software system which has resulted from the software

development process. These last two activities are exclusively carried out by members of the software development organization, although they are frequently performed by people who were not involved in the previous activities of preparing software requirements and design. When the system has been tested "successfully", it is released to the users and enters an "operational" phase. To the programmers who must work with the system, this phase is more commonly known as the "maintenance" phase.

The software maintenance phase is in some sense a repetition of the activities of the software development process. Although maintenance objectives include improving software performance, correcting errors, transferring software systems to new computer system configurations and deleting obsolete features, the most frequent objective is to increase system functionality by adding new features or by improving existing features. Thus, it is again necessary to discuss the requirements for the software system with the users; it is again necessary to perform software design; and, finally, it is again necessary to perform coding and testing. However, there is one fundamental difference in these activities when they are carried out during the software maintenance phase: these activities must now be carried out in the context of an existing, operational software system. It is important for the

software maintenance personnel to have an understanding of the process which was used to develop the software system. They must know not only what the operational system is and does, but also how and why it does so, since they will have to change the requirements, redesign the software, modify the programs and test the new implementation based on various demands. The traditional approach to providing information to assist with these tasks is by means of "system documentation". Many techniques have been developed to document software systems, but they tend to be incompatible and not sufficiently comprehensive to describe the entire software development process (e.g. HIPO [STAY76]). We have developed a model which is suitable for describing software systems' requirements, designs and programs. In addition, this model also permits individual software requirements to be traced through the intermediate levels of software design to the final programs of the system. This tracing capability is essential for the maintainer of a software system, who must be able to understand and modify the system rapidly and correctly.

Although it is important to identify the correspondence between the requirements which are to be changed and the code which must be changed as a result, there are several tasks which must be performed by the maintenance personnel before the modified software system can be made operational again. These

tasks constitute our software maintenance methodology [YAU78, 80a, 80e, 82c], and they are shown in Figure 2.2. After determining which parts of the software system must be changed in order to affect the modification request, software changes must actually be carried out, their consequences must be analyzed, and the modified system must be retested.

In the following sections we will describe our approaches to each of these problems of software maintenance. In Section 3, we will describe a software system model which may be used to trace the correspondence between the software requirements, software designs and programs of large-scale software systems. In Section 4, we will then summarize our approach for improving the reliability with which the program code can be modified - using a program slicer to assist in locating the code to be modified and a structure-oriented editor to make the modifications free from syntax errors. In Section 5, we will summarize our ripple effect analysis technique, which is used to analyze the effects of the program modifications on the behavior of the program. This static analysis technique allows potential logical and performance changes to be identified. The final phase of our methodology is to retest the modified system. In Section 6, we will summarize our module testing technique, which reuses existing test cases whenever possible to reduce the retesting effort.

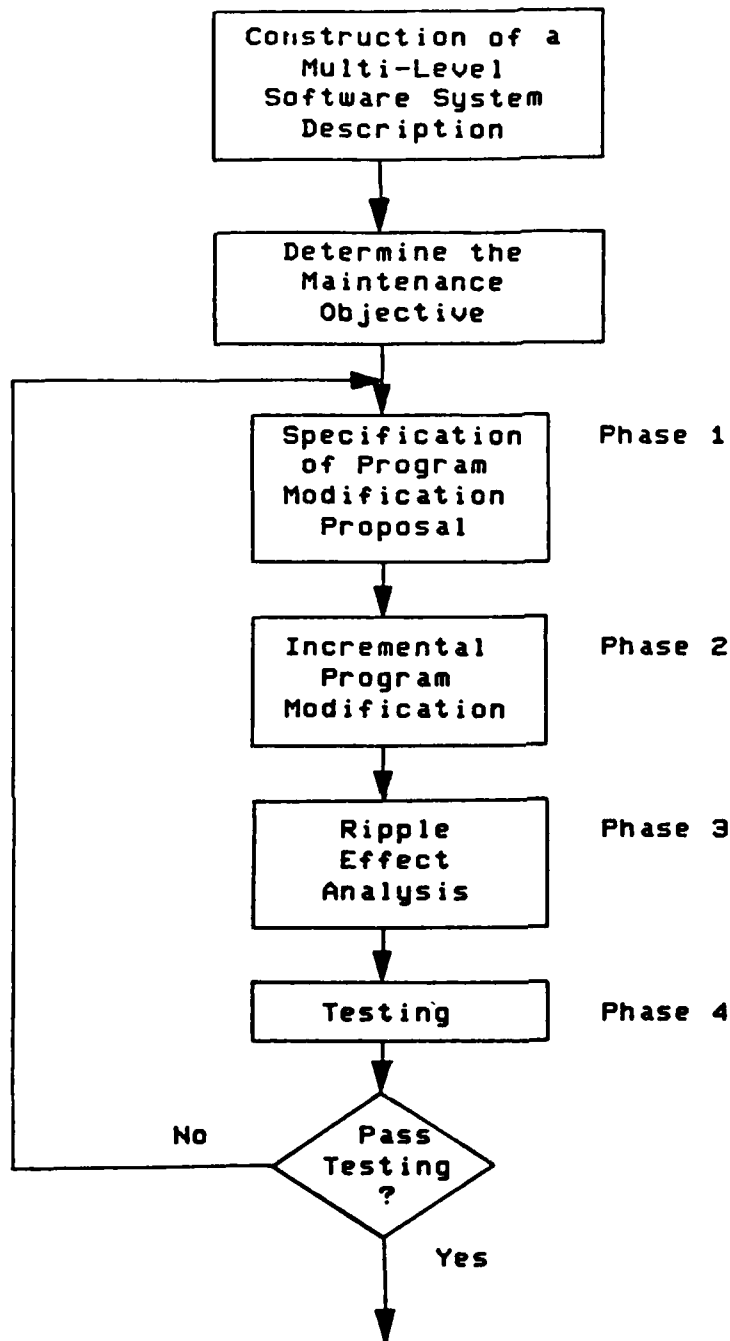


Figure 2.2. The software maintenance methodology

2.2 Tools And Techniques For A Software Maintenance Environment

The techniques for realizing program modification proposals, ripple effect analysis and module testing have been demonstrated by implemented programs running on a DEC VAX-11/780 computer under the VMS operating system. The technique for defining and tracing the correspondence from software requirements, via software design to program code was not implemented during the time available.

Tools for software maintenance should be able to share a common program representation. This integration of tools provides maintenance programmers with a standardized environment for performing maintenance activities. We have developed a formal program representation to support the tools described in Section 4, which permits an efficient implementation of our tools for program modification. In addition, we have developed efficient representations of programs for implementing each of the ripple effect analysis techniques. While this approach to implementing software tools is sufficient to demonstrate the validity of individual techniques, software tools based on these techniques will be of greater practical value if they share a common model of the program. A more flexible program model, such as the

Hierarchical Graph model [YAU80d, 81a, 82b], provides the means for combining different software tools into an integrated software maintenance environment. Like the model used by our syntax directed editor, this model is based on the abstract parse tree of programs. Since it also includes detailed information about data flows in the program, it appears to provide a suitable basis for integrating our individual software maintenance techniques into a set of practical, cooperating tools.

The ripple effect analysis techniques have been developed to perform exhaustive analysis in the sense that they are capable of identifying all blocks of a program which may be affected by a program modification [YAU78, 80a, 80b, 80c] [HSIE82]. However, to implement such a technique as a practical tool requires that we allow the maintenance programmer to restrict the tracing of ripple effects in accordance with his/her own understanding of the software system. Our implementation of the the logical ripple effect analysis technique permits the programmer to interact with the analysis program to select certain procedures for analysis and to remove others from consideration. Additional effort on the interface to these tools would be needed to improve their practical effectiveness.

Although our work has been to develop techniques for software maintenance, they are also useful during certain stages of software development. Our approach to realizing software modification proposals, for example, uses a syntax directed editor - a tool which is also very useful for the initial writing and debugging of programs. Furthermore, the activities involved in debugging a program require the identification of two types of code: the first may cause certain unintended effects (bugs), the second may be affected because of changes made to repair bugs. However, the program slicer (Section 4) has been developed to identify code of the first type, while ripple effect analysis (Section 5) is intended to identify code of the second type. In practice, we would expect these tools to be used even more effectively in the development phase, since the development programmer can take advantage of his/her familiarity with the program under development.

2.3 Quality Factors Affecting Software Maintenance

One important concept which runs throughout the entire software maintenance methodology is the use of software metrics. Our long term goal is to develop a software metric for modifiability - to provide a quantitative indicator of the amount of effort required to make changes to particular

programs or modules, and we have already developed some measures of certain attributes of modifiability, which will be described in detail in Section 7. The earliest measures which we have developed are those for the logical stability of programs and modules [YAUB0e]. These are based on our ripple effect analysis technique, and have been proposed as indicators of the resistance of a program or module to ripple effects as a result of changes made to it. We have also developed a measure for the logical stability of program design [YAUB2c] since we recognize the value of an early indication of deficiencies in the quality of a software system. However, a metric will not really be useful until it has been shown to correlate with the phenomenon which it is supposed to measure. We have, therefore, devoted some additional effort to the validation of our proposed stability metrics, and the preliminary results of our validation experiments will also be presented in this report.

3.0 SPECIFICATION OF SOFTWARE MAINTENANCE PROPOSALS

As reported by Lientz and Swanson [LIEN80], the most frequent and most costly activity under the heading "software maintenance" is system enhancement in response to user requests for change. The view of a software system as seen by its users is shown in Figure 3.1.

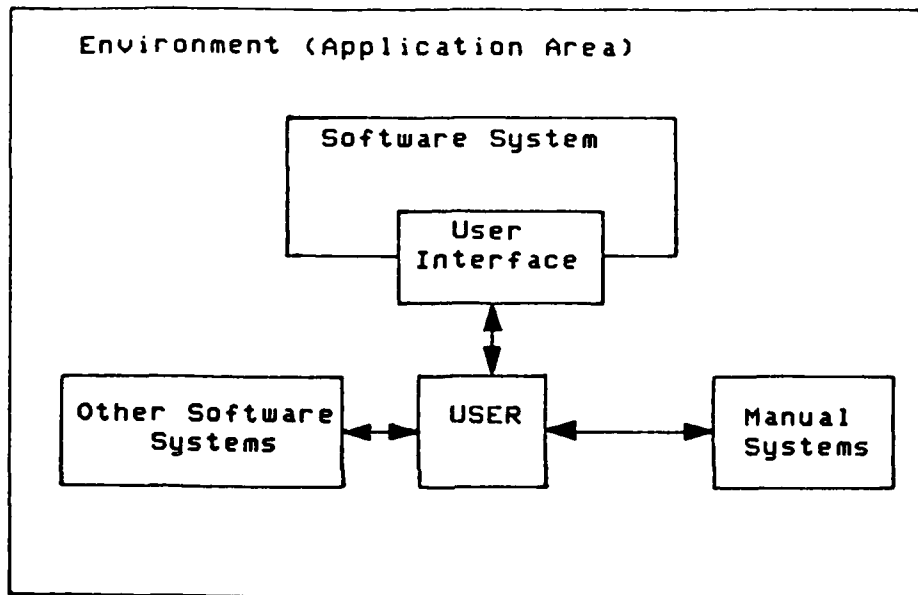


Figure 3.1. The users' view of a software system.

The users know the application area - the environment in which the system operates - and they know something about what the system requires as input, and what it is capable of producing as output. However, it is only rarely the case that they know anything about the internal organization of the system. Under these circumstances, user requests for change are inevitably stated with reference to the application area.

These requests usually refer to the interface which already exists between the software system and its operating environment. It is with such change requests that the process of specifying software maintenance proposals begins.

In order to correctly modify a software system, it is necessary to understand the relationship between the change requests and the programs which make up that system. Since this requires a clear understanding of both the behavior of those programs and the effects of the requested changes, a vast amount of effort or prior experience with the system is necessary. In the absence of such effort or experience, the most logical alternative is to record information which describes the relationships between the program code and the software system's application area.

The programmers' view of the same system is shown in Figure 3.2. During the maintenance process, these two views of the same software system (the users' view and the programmers' view) must be reconciled in such a way that the enhancements requested by the user are implemented. This requires changes to be made in both the users' and the programmers' views of the system and these changes must continue to be compatible with each other.

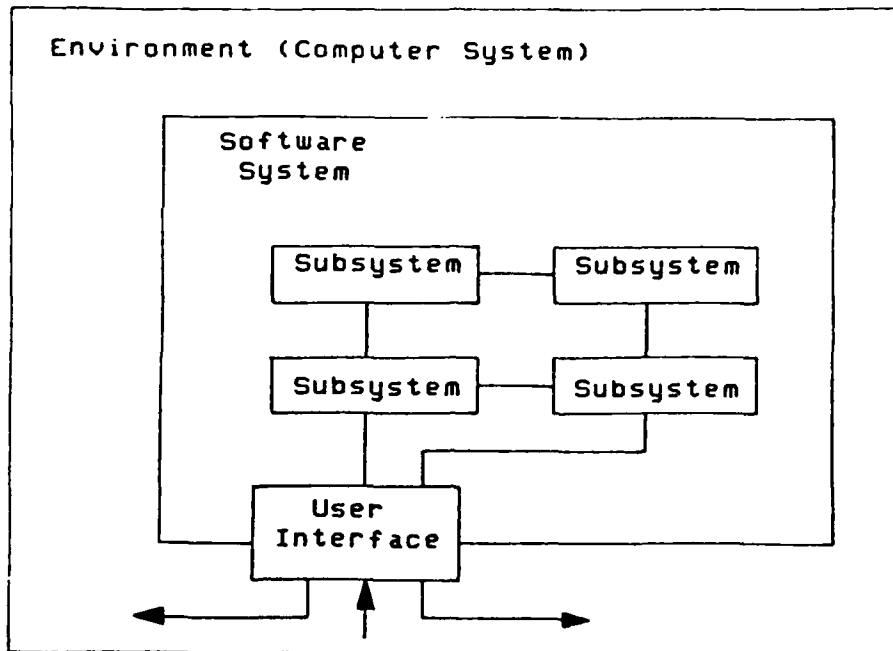


Figure 3.2. The programmers' view of the same software system.

Figure 3.3 shows the "semantic equivalence" relationship which exists between the two views of the original software system, and which must be preserved during the maintenance process. In addition, the users' new view of the system must represent the incorporation of the modification request into their old view of the system. In order to ensure this, the modification proposal which is implemented by the maintenance

programmers must preserve a "semantic equivalence" between the users' new view and the programmers' new view.

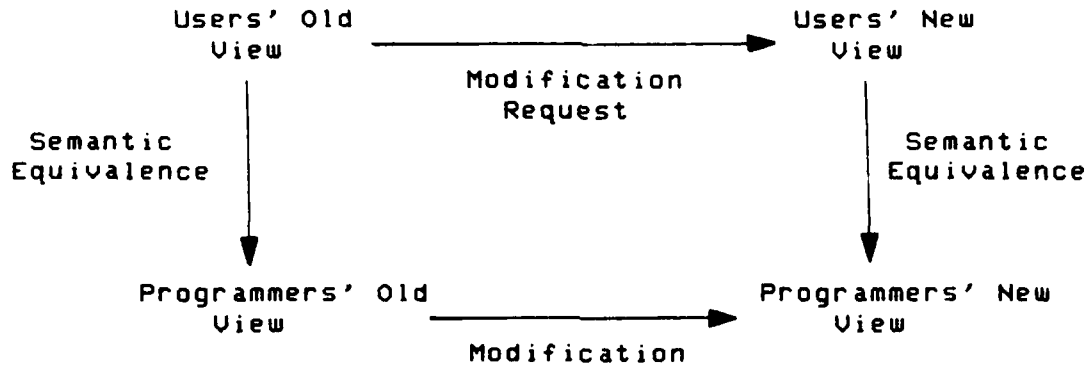


Figure 3.3. Equivalence preserving requirements for reliable software modification.

So far we have only discussed the problem of specifying a software maintenance proposal in a very abstract manner. Now, we would like to consider some of the practical problems, especially those involved in providing automated assistance for the maintenance personnel who must make the "semantic equivalence preserving" modification.

The first problem is to describe software systems using formal notations or formal descriptions. Since we cannot expect any automated assistance in dealing with informal notations or descriptions, we must ensure that all notations used to describe the software system have been formalized as much as possible. In dealing with the programmers' view of the

system, we are on fairly solid ground with respect to formal notation, since all programming languages must at least have a well-defined syntax - to allow automatic elimination of some programs which are clearly incorrect. In addition, all programming languages must have a semantic definition so that the programmer can predict the behavior of the code being written. However, these semantic definitions are frequently informal, are often subject to implementation constraints and occasionally permit several interpretations. When dealing with the users' view of a software system, we cannot expect that a very formal notation is being used. The best we can hope for is that parts of the system have been defined in a notation such as RSL [ALF077], SADT [ROSS77] or SA [DEMA78], which have varying degrees of formalization. If we do not have such a description of the system, one must be developed; otherwise, we will be unable to have any precise idea of what a change request entails until we have found the relevant program code which must be changed. One major problem with this approach is the likely existence of (though perhaps minor) discrepancies between the users' actual concept of the system operation and the programmers' description of that concept. However, given formal descriptions of these two views of the software system, we can proceed to study the effects on the one of changes made to the other. In order to deal with these issues, we will develop formal models of the different views of the software

system and proceed by working with these models.

3.1 A Model Of Software Systems For Software Maintenance

The most important questions to answer when we decide to model the processes and products of software maintenance are what to model and how to model it. We now describe how we have approached these problems, and explain the reasons for our choices. We will then present the details of our modelling approach.

3.1.1 Background

The major activity of the software maintenance process is to make changes to existing documents which describe a software system. These changes may be trivial or substantial, optional or essential. They may be carried out by a single person or by several independent groups of people. Since these documents are interdependent (for example, the design document is derived from the requirements document), we must also be able to model the process of changing a document in response to changes in another document. It is frequently necessary to retain several versions of each document, and therefore we must also control modifications so that they are made to the correct version and in the correct sequence. Thus, we have identified the following three major activities for which our model is needed:

- 1) Modifications to a single software document, by either (a) a single programmer or (b) several independent programming groups.
- 2) Replacement of portions of a software document in response to changes made to its source document.
- 3) Control of different versions of individual documents and their interdependencies.

In practice, the software documents which must be modified may exist in either a textual or graphical form. However, in both cases there is a substantial amount of context sensitive information present in the document. Due to the limitations of the descriptive power of strings (and even trees) when modelling context sensitive information, some other approach is required. Therefore, we have chosen to use graphs, with their greater descriptive power, to directly show context sensitive properties.

Having adopted the graph as the basic representation for software documents, we must express changes to these documents as graph modifications. Graph modifications are commonly described by means of graph rewriting systems. Using existing methods for studying graph rewriting, it is possible to control concurrent access to a software document, since tests have been

developed to check if two separate modifications to a single graph are sequential independent (may be executed in either sequence) or parallel independent (may be executed concurrently). These checks are necessary when several groups work together to modify a large software system. When the modifications are interdependent, these tests may be used to identify the interface (or interaction) region of the two modifications on the graph.

3.1.2 Graph Rewriting Systems

Graph rewriting systems have become a topic for research in recent years [CLAU79], primarily as a result of the great significance which graphs and graph theoretic concepts have assumed in computer science and engineering. Since we wish to model the processes of software maintenance, and to do so in a very abstract manner, it is natural to examine the use of such an abstract tool, particularly in view of the preponderance of graph representations for software requirements and design.

3.1.2.1 Graph Rewriting

To rewrite a graph means that we will apply a set of rewriting rules to the graph, one by one, in some sequence, to construct another graph. A rewriting rule corresponds so closely to a production rule of a grammar for a language that

graph rewriting systems are also known as "graph grammars".

Each graph rewriting rule has a left-hand side and a right-hand side, each of which is a graph. To apply a rewriting rule with the left-hand side L and the right-hand side R to a graph G , it is first necessary to locate an instance of the graph L as a subgraph of G . If no such instance exists, then the rewriting rule cannot be applied. If such an instance does exist, then it should be (conceptually) deleted from G , giving rise to the graph $G - L$, and then the graph R should be (conceptually) added to G in its place, giving rise to a new graph $H = (G - L) + R$.

The most difficult part of the entire process is to embed the right-hand graph R into G in place of L . When strings are being rewritten, this embedding of the right-hand side is made obvious by the implicit left to right ordering of the characters in the string. This is illustrated in Figure 3.4.

In Figure 3.5 we show the difficulty involved in embedding a graph within a graph. The rewriting rule shown there requires that we replace the node labelled "a" by a subgraph consisting of three nodes (labelled "b", "c" and "d") and two arcs (from "b" to "c" and from "c" to "d"). Clearly, the rewritten graph must contain five nodes, labelled "b", "c", "d", "e" and "f". In addition, it must contain arcs from "b"

Rule

a => bcd (Replace "a" by "bcd")

Applications

If the original string is "baabe" then the following applications of the rule may be made.

- 1) baabe => baaabe => bbcdaabe => bbcdabe
- 2) bbcdabe => bbcdabe => bbcdbbcdbe => bbcdbbcdbe

Figure 3.4. An example for rewriting a string.

to "c" and from "c" to "d". However, what should be done with the arcs in the original graph from "a" to "e" and from "a" to "f"? That is, how should the new subgraph be embedded into the original graph? The approach which we have adopted is to assign integer labels to certain nodes or arcs in the rewriting rule, with the constraint that any integer which appears on the left-hand side of the rule must also appear on the right-hand side. The interpretation of this assignment of labels is that when a rule is applied, any node labelled "i" on the left-hand side is considered to be replaced by the node labelled "i" on the right-hand side so that any arcs incident to (or from) that node in the original graph should be incident to (or from) the replacement for that node in the graph on the right-hand side. In Figure 3.5, there is only one node to be replaced (labelled "a") and its replacement node is the node labelled "c" (as shown by the integer label "1"). Thus, the rewritten graph is

the one shown at the bottom of that figure.

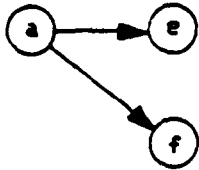
Rule



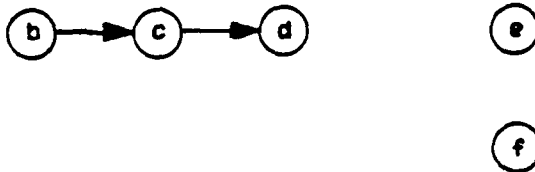
(Replace node "a" by a graph with nodes "b", "c" and "d")

Applications

If the original graph is



then the following (partial) application of the rule may be made.



The final (rewritten) graph is

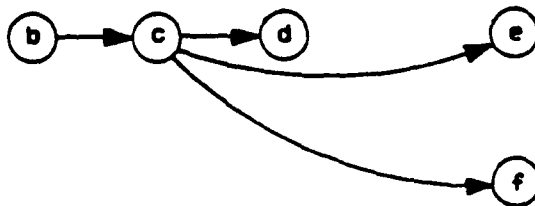


Figure 3.5. An example for rewriting a graph.

3.1.2.2 Definition Of A Labelled Graph

To formalize the graph representations for software systems, we define a labelled graph as follows: A labelled graph is an 8-tuple,

$$G = (N, A, LN, LA, sN, tN, nL, aL),$$

where N is a set of nodes,

A is a set of arcs,

LN is a set of node labels,

LA is a set of arc labels,

$sN, tN : A \rightarrow N$ are functions which map each arc to its source and target nodes (respectively),

$nL : N \rightarrow LN$ is a function which maps each node to its label,

$aL : A \rightarrow LA$ is a function which maps each arc to its label.

3.1.3 The Intraphase Model

The software documents used to describe each phase of the software development process will each be modelled by a set of interconnected components of the software system. We represent each component by its control flow, data flow and data structures, and also by its relationships to other components. Its interface with other components is stated in terms of objects required from other components and objects provided for

other components. A software system is simply a collection of such components, with a distinguished initial (or master) component.

3.1.3.1 Software Components

A software component is an executable object which contains several subcomponents. These subcomponents are:

- a control flow structure (in a form to be described),
- a set of data structure graphs (of a similar form),
- a set of data flow triples, whose executable objects are "leaves" of the control flow graph and whose (input and output) data objects are data structure graphs, and
- a set of distinct object names, each of which refers to a data structure graph or module which defines the structure of that object.

3.1.3.1.1 Control Flow

The following notation will be used to describe the control flow of a software system component. Since it emphasizes only the relative ordering of activities, this notation is independent of the particular notation being used to describe the component. We have confirmed that it can be used to describe most of the control flow properties of a requirements definition in RSL or a program in PASCAL. This notation uses the formalism of a labelled graph, using nodes to

represent "activities" and arcs to represent relationships between these.

Let us now introduce our notation. First of all we specify the basic notation completely, and then describe the remainder of the notation informally.

A basic, structured, sequential control flow description is a labelled graph with

$$LN = \{TASK, LOOP, AND, OR\} \cup Z^+ \cup \{e\},$$

$$LA = Z^+ \cup \{e\},$$

where Z^+ denotes the positive integers $\{1, 2, 3, \dots\}$ and e denotes the empty string.

The use of these symbols is now explained informally: The graph is a rooted tree structure, directed downwards. Nodes labelled by LOOP, AND or OR are referred to as structured nodes. E-labelled nodes are referred to as primitive nodes. Nodes and arcs labelled by e are referred to as e-labelled. Nodes and arcs labelled by a positive integer are called Z-labelled. Structured nodes are always nonterminal nodes in the tree. Primitive nodes are always terminal nodes (leaves) of the tree. All leaves of the tree are e-labelled. All nodes and arcs of the tree are labelled by a label from LN or LA.

1) Form: The software component has a single, distinguished node, labelled TASK. This node is the root of the tree.

Interpretation: The subtree of which this node is the root represents a separately defined, executable software component.

2) Form: A LOOP node always has a single child.

Interpretation: The activity represented by the subtree rooted at the child of the LOOP node is to be executed a number of times, ranging from zero to a finite number to be decided within the LOOP node in an (as yet) unspecified manner.

3) Form: An AND or OR node always has a single child, which must be an Z-labelled node.

Interpretation: An AND node indicates that the children of its Z-labelled child must all be executed, while an OR node indicates that one child of its Z-labelled child must be executed.

4) Form: Any Z-labelled node, with label n , must also have outdegree n , and its parent in the tree must be labelled by either AND or OR. The arcs of which this node is the source must be labelled by the positive integers $\{1, 2, 3, \dots, n\}$.

Interpretation: The value of the arc label indicates the order in which the activity should be executed. The activity labelled i should be executed before the activity labelled $i+1$.

In summary, AND represents the execution of a sequence of (n) activities, OR indicates the selection of 1 (of n) activities, and LOOP represents the repeated execution of an activity.

3.1.3.1.1.1 Conditional Expressions

Our current approach to the expressions which control selections and iterations is to restrict them to be of one of two types: they may have the form of either a range of values or a condition (or boolean expression). Figure 3.6 shows an example of an RSL statement and its graph representation.

RSL statement

```
IF FOUND = TRUE  
  ALPHA: A1  
OTHERWISE  
  ALPHA: A2  
END
```

Control flow representation

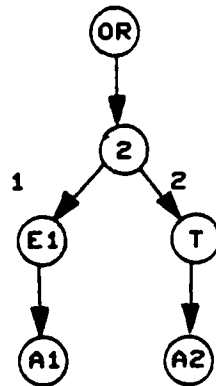


Figure 3.6. An example of the control flow representation.

3.1.3.1.1.2 Notational Extension

As a notational convenience, we may represent the tree structured graphs in the following manner. This simple convenience presents the graphs which make up the model in a more pleasing manner. We will use abbreviations for the standard graph patterns as shown in Figure 3.7. The control flow representation given in the previous section would appear as shown in Figure 3.8.

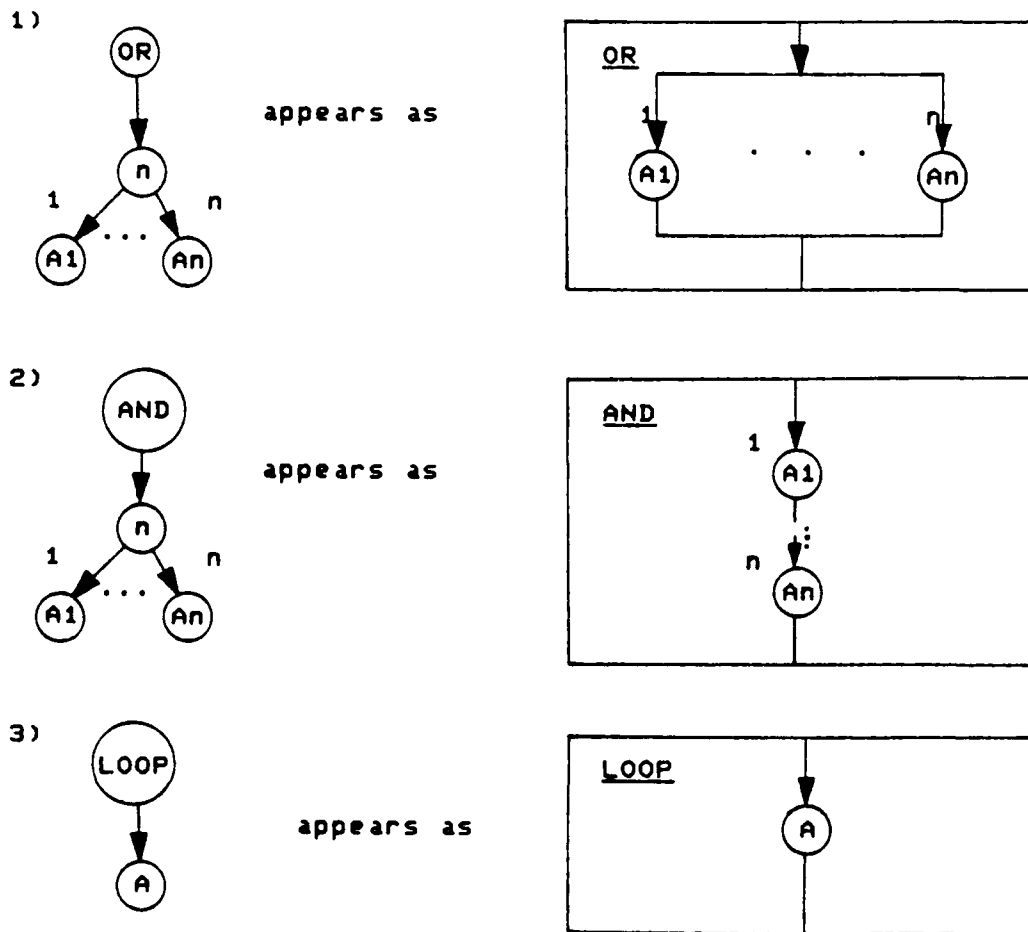


Figure 3.7. Notational abbreviations for the standard graph patterns.

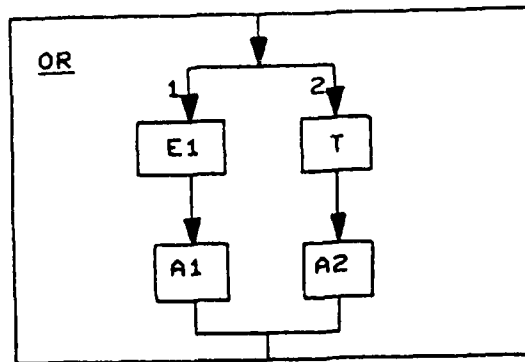


Figure 3.8. Abbreviated representation of the example shown in Figure 3.6.

3.1.3.1.1.3 Extensions To Other Control Structures

Extensions to this basic form of control flow description have been defined to describe (unstructured) jumps and concurrency or nondeterminism. Jumps are included as directed arcs between two nodes, the arc being specially labelled to distinguish it from the arcs representing structured control flow. Concurrency or nondeterminism are included by permitting Z-labelled nodes to be the source of e-labelled arcs. This removes the ordering concept described in Form 4) discussed before, and so permits nondeterminism. A further extension has been defined to support inclusion of separately defined software components within another component. This represents both the SUBNET concept of RSL and the procedure concept of programming languages, such as PASCAL. With these extensions the graph is no longer a tree structure, but the non-tree arcs are distinctively labelled.

Using such an abstract view of control flow, it is possible to construct, for example, the skeleton of a PASCAL program from the control flow requirements of an RSL specification. In addition, the theory of graph modification [CLAU79] provides us with a foundation for defining modifications formally, and for relating this formal definition to modifications which are to be made to software systems' descriptions in notations which are currently in use.

3.1.3.1.2 Data Flow

Data flow information has also been added to our model. This information may be viewed as a set of triples of the form:

$$\langle EO, IO, OO \rangle$$

where EO is an executable object (such as a statement or procedure), and IO (the input object) and OO (the output object) are data objects (such as program variables). Such a triple has the interpretation that EO may use the value of IO to alter the value of OO. In its graphical form, each such triple denotes the existence of an arc from activity EO, labelled OO, to some other activity, and from some other activity to activity EO, labelled IO.

For example, the activity A1, written as an ALPHA in RSL, appears as:

ALPHA: A1.
INPUTS: DATA: D1.
OUTPUTS: DATA: D2
 DATA: D3.

and would be represented as:

<A1, D1, D2>
<A1, D1, D3>

unless further information is available. However, if we have information that D3 is being assigned a value in A1 which is independent of D1, then we would represent A1 as:

<A1, D1, D2>
<A1, K, D3>

where K is some relevant constant or other independently defined data object. Figure 3.9 shows the graphical representation of this latter case.

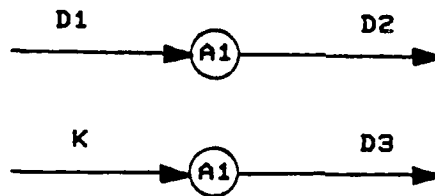


Figure 3.9. An example of the data flow representation.

3.1.3.1.3 Data Structures

In the previous section we described a graph representation for control flow, and indicated that we have also developed a very similar representation for data structures. We will use our data flow information to connect representations of data structures (which we call "input data objects" or "output data objects") to representations of control flow structures (which we call "executable objects"). The graph representation of data structures resembles that used to describe executable activities, in that sequences of heterogeneous data objects are represented by trees rooted with an AND node, selections of one of several data objects are represented by trees rooted with an OR node, while collections of several homogeneous objects are represented by trees rooted with a LOOP node. For example, the data item D1, written in RSL as:

```
DATA: D1.  
  INCLUDES: DATA: D1-P1  
            DATA: D1-P2  
            DATA: D1-P3.
```

would be represented as shown in Figure 3.10. In the event that the subcomponents of D1 are also structures, then their structure will also become a substructure of D1.

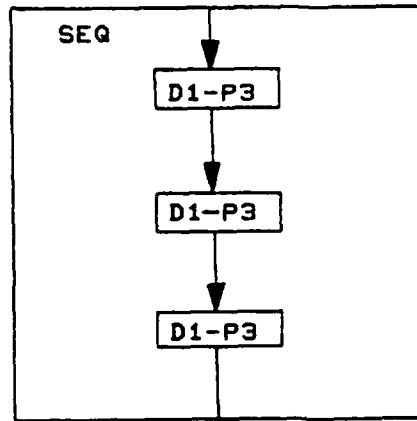


Figure 3.10. An example of data structure representation.

3.1.3.1.4 Data Dictionary

Within the description of each component is a data dictionary. As is customary, this dictionary contains a definition of each element of this component, excluding those which belong to other components, but are used within this component. There are three types of elements which exist in any component - activities, data and structures.

Activities are defined by the data items which enter them or leave them. They also describe the operations which are performed on that data. These operations include operations defined by the language, notation or operating environment, and those carried out by other components of the system. Examples are the "+" operation of a PASCAL program, which probably refers to a hardware dependent addition instruction, and the

PASCAL "sin" (sine of an angle) function, which probably refers to a function in the system's standard library of functions.

Data are defined only by their structure and external name. The structure of a data item may be defined by the language, notation or (less often, but hardware dependent) operating environment, or by other components of the system. Examples are the standard file "input" of a PASCAL program, which refers to a standard system input file (usually the terminal keyboard), and the PASCAL constant "maxint", which has the value of the largest integer available to PASCAL programs in a particular computer system.

Structures are defined in terms of connected sub-components, which are themselves either data items or other structures. Subcomponents may refer to structures defined by the language, notation or (less often, but hardware dependent) operating environment, or by other components of the system. Examples are the standard types "text" and "integer" of PASCAL programs. "Text" refers to the system's implementation of sequential files of characters, while "integer" is affected by the available word length and precision of the computer system.

The data dictionary is a sub-structure indexed by an internal object name, denoting a particular activity, structure or piece of information. When the object is a named activity,

the index leads to another software component. When the object is an unnamed activity, the index leads to a description of the activity (which may be in a formal or informal notation). When the object is a structure, the index leads to a description of the form of that structure. The lowest level structures are those defined by the computer installation. When the object is a piece of information, the index leads to the definition of the structure which is contained in the piece of information.

For instance, the previous examples would give rise to the following data dictionary entries:

<D1, DS1> where DS1 is the structure shown in Figure 3.10,
<D1-P1, DS2>
<D1-P2, DS2>
<DS-P3, DS2>
<A1, CS1> where CS1 is the control flow structure shown in Figure 3.8.

Furthermore, DS1 and DS2 are the names of data structures to be found within other components, and D2, D3 and K are also assumed to be defined within other components.

3.1.3.2 Component Interfaces

Any software component is a separately defined activity in an overall software system. In order to act in a coordinated manner, the components must share information with each other and provide services for each other. We would like to represent the interdependencies between components in a

disciplined fashion, in a way that permits modifications to be analyzed and to match the representation of software components. Our approach to this problem is to associate an interface subcomponent with each software component. Within this interface are defined all of the objects which appeared outside the current component, and also all of the objects which appeared inside the current component, but may be used by other components. These objects are further distinguished between those which are directly linked to external components and those which are indirectly linked (as parameters).

The interface graph can be formally defined as follows:

An interface graph is a labelled graph :

$$G = (N, A, LN, LA, sN, tN, nL, aL)$$

where

$$LN = \{INTERFACE, GLOBALS, PARAMETERS, IMPORTS, EXPORTS\}$$

$$U \mathbb{Z}^+ \cup \{e\}$$

and $LA = \{e\}$

The graph is a rooted, acyclic, directed graph (acyclic "digraph"). Nodes labelled by GLOBALS, PARAMETERS, IMPORTS and EXPORTS are referred to as structured nodes. Nodes and arcs labelled by e are referred to as e-labelled. E-labelled nodes are referred to as primitive nodes. Nodes and arcs labelled by a positive integer are called Z-labelled. Structured nodes are always nonterminal nodes in the tree. Primitive nodes are

always terminal nodes (leaves) of the tree. All leaves of the tree are e-labelled. All nodes and arcs of the tree are labelled by a label from LN or LA.

1) Form: The component interface has five distinguished nodes, labelled INTERFACE, GLOBALS, PARAMETERS, IMPORTS and EXPORTS. The INTERFACE labelled node is the root of the digraph. The other four distinguished nodes are immediate descendants of the root node. No other node is directly connected to the root node

Interpretation: The subgraph of the root node represents the interface between this component and other components. All references to or from other components are forced to pass through this subgraph. This subgraph includes all names and structural information which is required to complete the interface with any external component.

2) Form: The primitive nodes of the interface digraph are directly connected to exactly one of the following nodes (GLOBALS, PARAMETERS).

Interpretation: The primitive nodes represent the interface objects. If a node is connected to GLOBALS, then the object represented by that node may be accessed directly. If a node is (instead) connected to PARAMETERS, then the local object represented by that node provides indirect access to another object, and this relation between the local object and the

other object may be altered by execution of the system of components.

3) Form: The primitive nodes of the interface digraph are directly connected to at least one of the following nodes (IMPORTS, EXPORTS) (connection to both of these is possible).

Interpretation: The primitive nodes represent the interface objects. If a node is connected to IMPORTS, then the object represented by that node may be examined and used, but may not be altered. If a node is connected to EXPORTS, then the object represented by that node may be altered.

For the previous example, the interface must include all of those objects which did not appear in the local data dictionary, i.e. DS1, DS2, D2, D3 and K. Since RSL will not permit data structures to be altered within the system, DS1 and DS2 must be connected to the IMPORTS node alone. In addition, since RSL has no facility for parameter passing, all five objects must be connected to the GLOBALS node alone. The data objects, D2 and D3, are both altered within the component, and hence they should be connected to the EXPORTS node alone. The constant K is defined externally, and cannot be altered in this component, and therefore it should be connected to the IMPORTS node alone. The interface graph is shown in Figure 3.11.

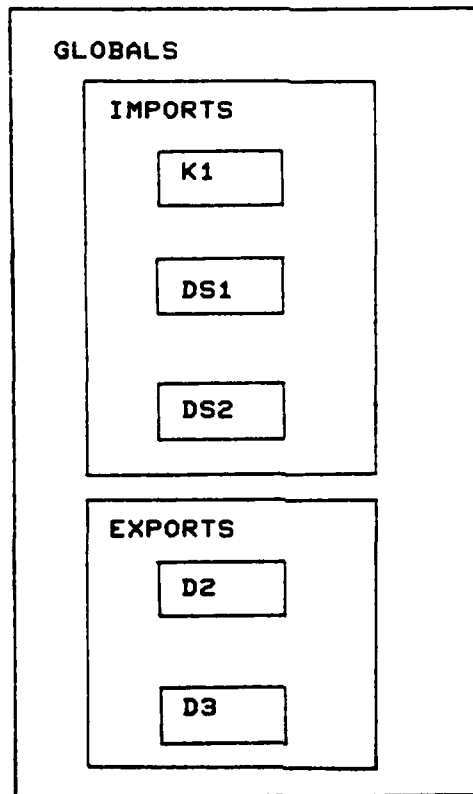


Figure 3.11 An example of a component interface graph with no PARAMETERS subcomponent.

3.1.4 The Interphase Model

Although our comparison between the features of different languages (for writing programs, designs and specifications) and the abilities of our model to represent such features has shown that we still have many limitations (e.g. no scope rules, no parameter passing rules), we think that our current model is suitably complete for us to study properties of interest when software modifications are proposed.

We now develop a method for expressing "equivalence" relationships between objects in two different levels of description. We have already indicated that these relationships should be expressed as relationships between graph structures, and have made a preliminary study of methods to achieve this. It is these equivalence relationships that provide us with the ability to analyze the effects of modifications to one level of description on the behavior of the other.

The interphase model is composed of a set of graph rewriting rules described before. On the left-hand side of each rule is a subgraph of the graph model of the software system at the end of a particular phase. On the right-hand side of each rule is a subgraph of the graph model of the system at the end of the next phase. The rules record the fact that the subgraph on the left-hand side is to be replaced by the subgraph on the right-hand side. Thus, if a change is made to a particular part of a software system, we can identify its potential impact on the next phase by locating all rules with that part of the system in its left-hand side, and identifying all of the subgraphs in the corresponding right-hand sides.

In order to constrain possible ripple effects, an effective restriction on the software development process is to require that each node of a graph model of a software system should appear on the left-hand side of exactly one rule. The effect of adopting this rule is to create a process closely resembling the "stepwise refinement" process advocated by many authors (e.g. [WIRT71]), but applied to the refinement also of data flows and data structures, whereas stepwise refinement is involved primarily with control flow and executable activities. For example, given a process structure, using the notation of Jackson's design methodology [JACK75] shown in Figure 3.12, we have the model representation shown in Figure 3.13.

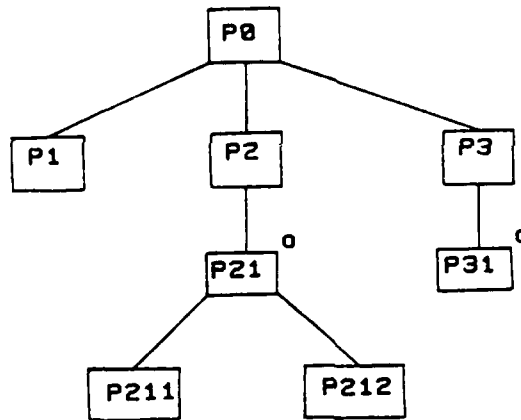


Figure 3.12 A process structure using Jackson's design methodology

In this case, the interphase model would contain the rules shown in Figure 3.14. These rules show how the design structure may be derived from the requirements structure. Of

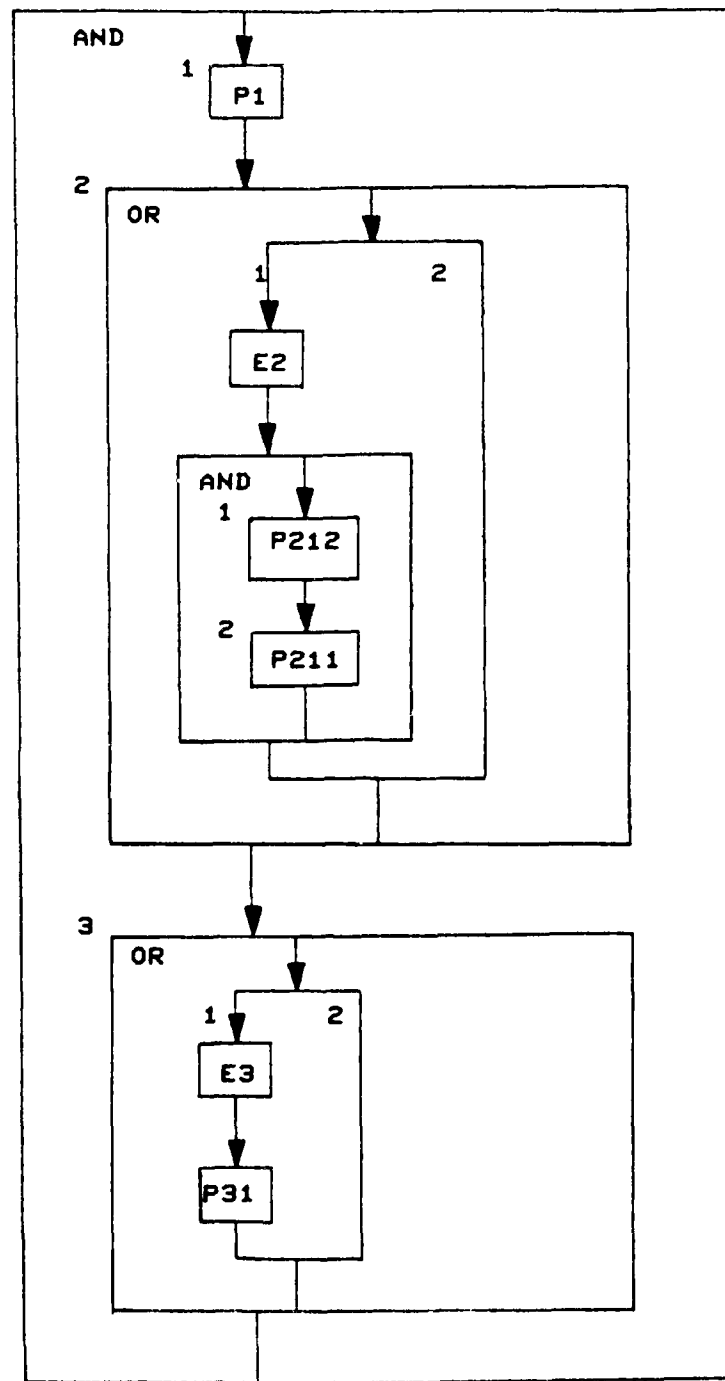


Figure 3.13 The model representation of the JDM design shown in Figure 3.12.

course, other derivations are possible, and the record should show the derivation that was actually used.

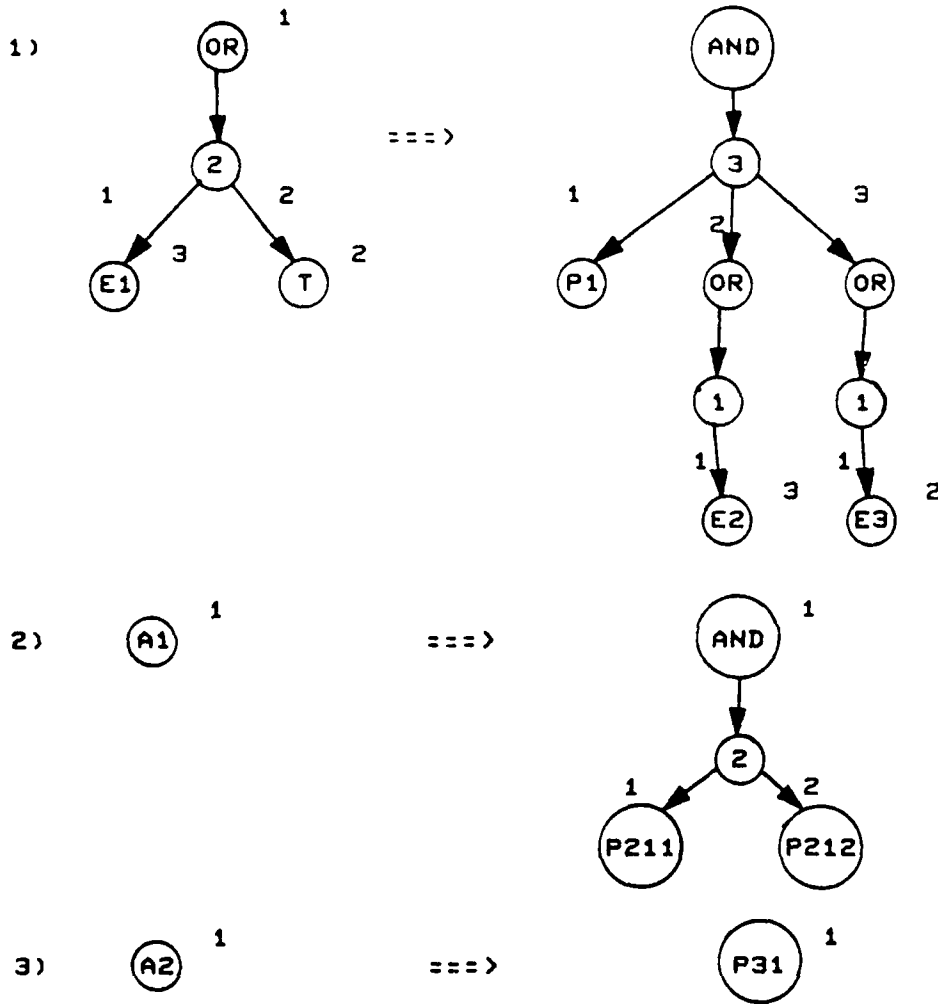


Figure 3.14. Part of the interphase model between requirements and design.

3.2 Construction Of The Software Model

In this section we will present a general technique to enable the software model in the format described above to be constructed for any software system. We will first describe the approach for constructing the model for a particular phase, and then the approach for constructing the interphase model.

3.2.1 Construction Of The Intraphase Model

The intraphase model describes control flow, data flow and data structure. Not every method of system documentation possesses all of these attributes. Nonetheless, these attributes, which refer to the sequence of activities, the flow of information and the form of information are present in all systems and relevant to all system descriptions.

Since we will use our model for many different notations to cover several phases, it is necessary to base the construction of our model on some properties which are independent of these individual notations. The properties chosen are based on semantic rather than syntactic properties, on the assumption that this basis will be sufficiently broad to support the general aims of the model. For this reason, construction of the intraphase model for a particular notation must be preceded by preparation of a semantic definition of

that notation.

Regarding the control flow, the execution sequence is emphasized. For example, let us consider the BNF production rule for the PASCAL syntax construct <compound statement>. The definition is:

```
<compound statement> ::=
  begin <statement> { ; <statement> } end
```

For the purposes of having a semantic model however, we are more interested in the fact that the list of statements is to be executed in their order of appearance than in the use of "begin", "end" and ";" as delimiting tokens of this construct. Therefore, we would "abstract away" from such a construct to give:

```
<compound statement> ::=
  <statement> { --> <statement> },
```

where $S_1 \text{ --> } S_2$ denotes that S_1 should be executed before S_2 .

Data flow properties are mainly associated with the assignment statements. For example, a PASCAL assignment statement and its data flow properties can be expressed in the following manner:

```
<assignment statement> ::= <variable> := <expression>
<variable> ::= <identifier> | <identifier> [ <index list> ]
<index list> ::= <index> { , <index> }
<index> ::= <expression>
```

```
AS ::= I --> O ( I = V.I U E.I; O = V.O U E.O )
V.O ::= {id} U IL.O, V.I ::= IL.I
IL.I ::= ind.I ( U ind.I ), IL.O ::= ind.O ( U ind.O )
```

ind.I ::= E.I, ind.O ::= E.O

To display data structure properties, we note that they come from the structure of declared objects. In PASCAL, for example, an array structure has the following representation:

```
<array declaration> ::=
  array [ <index range list> ] of <type>
<index range list> ::=
  <index range> { , <index range> }
```

with the interpretation:

```
<type>          <type>          . . .          <type>
  ^              ^                  ^
  |              |                  |
  |              |                  |
<index> -->(succ)-> <index> -->          . . .          -> <index>
```

Therefore, definition of the intraphase model for a particular notation is largely a manual procedure. The notation must be analyzed to identify the features determining the order of events, the flow of information and the form of information. These features of the notation must be characterized in terms of the basic elements of the model.

3.2.1.1 Definition Procedure

- 1) For each construct in the language definition which corresponds to a distinct activity, define an entity.

- 2) For each construct in the language definition which defines a sequencing relationship between other constructs, define a relational entity.
- 3) For each construct in the language definition which defines information flow into or out of an activity, define a triple $\langle \text{activity, I-set, O-set} \rangle$.
- 4) For each construct in the language definition which defines the form of a piece of information, define a $\langle \text{name, structure} \rangle$ pair, where $\langle \text{structure} \rangle$ is derived from the form of the information.

3.2.1.2 An Example For Constructing An Intraphase Model (An RSL Subset)

Here we describe the construction of a model for a subset of the Requirements Statement Languages (RSL) [ALF077]. In the following definition, the nonterminal symbols are delimited by "<" and ">", optional symbols are delimited by "[" and "]", choices to be made between symbols are delimited by ":" and symbols to be repeated are delimited by "(" and ")", with preceding and succeeding integers to denote the lower and upper bounds respectively on the number of iterations. The definition of the subset of the language now follows:

```
<new element definition> ::=
  [DEFINE] element-type-name element-name [comment].
  Ø( [INSERT] <element definition sentence> )n

<element definition sentence> ::=
  <attribute declaration>
  ; <relation declaration>
  ; <structure declaration>

<attribute declaration> ::=
  attribute-name 1( value-name : number : text-string )1
  [comment].

<relation declaration> ::=
  relation-name [relation-optional-word]
  1( [element-type-name] element-name [comment] )n.

<structure declaration> ::=
  STRUCTURE 2( <node> )n END [comment].

<node> ::=
  <element node>
  ; <terminator>
  ; <and node>
  ; <or node>
  ; <for-each node>

<element node> ::=
  [element-type-name] element-name [comment]

<terminator> ::=
  TERMINATE [comment]
  ; RETURN [comment]

<and node> ::=
  DO [comment] <branch>
  1( AND <branch> )n
  END

<branch> ::=
  1( <node> )n

<or node> ::=
  IF [comment] <conditional branch>
  Ø( OR <conditional branch> )n
  OTHERWISE [<branch>]
  END
```

```
<conditional branch> ::=  
  [unsigned-integer] <condition> <branch>
```

```
<for-each node> ::=  
  FOR EACH [FILE] file-name [RECORD]  
  [SUCH THAT <condition>]  
  DO [comment]  
    1{ [ALPHA] alpha-name [comment]  
      : [SUBNET] subnet-name [comment]  
    }1  
  END
```

```
<condition> ::= (<Boolean expression>)
```

Now, following the definition procedure given in the last section, we have the following steps:

- 1) Construct an entity for each ALPHA and SUBNET and STRUCTURE in the software system's requirements.
- 2) Construct relational entities for each <and node>, <terminator>, <or node> and <for-each node>.
- 3) Define triples <alpha-name, I-set, O-set>, <subnet-name, I-set, O-set> and <structure-name, I-set, O-set> for each ALPHA, SUBNET and STRUCTURE.
- 4) Define pairs <data item name, Structure> from the definitions for DATA items.

3.2.1.3 Implementation Of The Intraphase Model

Since the intraphase model is based on a semantic definition of a notation, it is clear that construction of the model should proceed from semantic analysis. Semantic analysis is most commonly carried out by a compiler (or an interpreter), in conjunction with a parser, which constructs the necessary syntax constructs with which semantic rules are associated. This is the approach which we will adopt for construction of the intraphase model.

To implement the intraphase model for a particular notation, the first step must be to develop a parser for that notation. Obviously this is only possible when the notation has been formally defined. If no formal definition is available, then one must be defined, or the parsing process must be replaced by a manual inspection process.

When we have developed a parser to recognize the notation in question, we will then modify it to produce the nodes and arcs of our model. The first step is to identify those constructs which constitute the primitive activities of the system from the definition of the notation. Now identify the syntax constructs which control the sequence of primitive activities and determine their effect on the activity sequence. Next, identify by what rules these activities can use or alter

the objects of the system. Now modify the parser in the following manner:

- 1) When the construct is a primitive activity, produce a "primitive activity" node.
- 2) When the construct is a program object, produce a "program object" node.
- 3) When the construct combines several activities, produce a "combinator" node and connect it to the primitive activities.
- 4) When the construct is a "callable" entity, produce a "component" node and connect it to its activities.
- 5) When the construct refers to a "callable" entity, produce a "primitive activity" node, but connect it to the "component" node.

3.2.2 Construction Of The Interphase Model

By the nature of the interphase model, it is clear that it depends on the notations used at each phase. Nonetheless, certain general principles provide general assistance in its construction.

The form of an interphase model is simply the form of a graph rewriting system, in which the previous intraphase model plays the part of the initial (axiom) graph. For each rewriting rule, the left-hand side must include nodes and arcs which are part of the axiom graph, while no right-hand side may include any node or arc from the axiom graph except that if the same node or arc also appears in the left-hand side of the rule.

The interphase model should be constructed by the software development team, and must be updated by software maintenance personnel who modify the system. In the event that no such model exists to support the software maintenance personnel, they must construct it from the existing documentation of the system. Heninger [HENI79] has described a successful maintenance project in which the software requirements for a complex flight control system were constructed by examination of existing documentation and discussion with users and developers of the system. The interphase model can be constructed by following that procedure and recording the relationships identified between the requirements derived and the code being examined.

Of course the simplest way to define graph rewriting rules to replace a graph G by another graph H is simply to use a single rule $G \Rightarrow H$. While this is both accurate and permissible, it is of little use to maintenance personnel because this is already an assumed rule, used by any maintenance programmer who trusts the software documents from which G and H were constructed.

On the other hand, if we define graph rewriting rules so that each left-hand side has exactly one node, then we are placing restrictions on the developer of H , since certain permissible graph structures cannot be constructed using this restriction [JANSBØ], although we are greatly assisting the maintenance programmer in performing tracing throughout the system. This phenomenon is further extended if we permit the node on the left-hand side to have only a single representative node on the right-hand side, by restricting the possible growth of node interconnections.

While it may appear to be undesirable to restrict the range of possible solutions available to the developer, a discipline in the use of development processes does assist the maintenance programmer. In addition, since the number of arcs in the graph is a measure of the degree of interconnection of the graph, it is an indicator of both the complexity and the stability of the system. Therefore, given McCabe's measure of

cyclomatic number for program complexity [MCCA76] and our experience with the effects of interconnectivity on program and design stability [YAUB0e, 82c], any development process which raises this degree of interconnectedness must be considered a source of increasing system complexity and instability.

3.2.2.1 Definition Of The Interphase Model

In order to construct an interphase model, it is necessary that two phase models already exist. We will refer to these as the source and target intraphase models, and we will say that the target model is derived from the source model. Having thus defined our terminology, we now state the definition procedure.

- 1) For each node in the source model, assign it to the left-hand side of one rule.
- 2) For each rule, assign a sub-graph of the target model to its right-hand side.
- 3) For each node in the left-hand side which is part of the interface between the subgraph and the complete graph model, indicate to the user any arcs by which it may be connected to the rest of the model. According to the user's response, select one of the following steps.
 - 3a) If the arc is not to appear in the next phase, the developer must enter an explanation for this omission.

3b) If the arc is to be represented in the next phase, the developer must identify the node or nodes of the right-hand side which "represent" the node under consideration. Each such node on the right-hand side should be given a unique label. The node on the left-hand side should be given a list of labels, made up of all of the labels which were just assigned to the right-hand side.

3c) If, in attempting to obey the instructions in the previous step, it is found that the arc is not represented by an arc or simple set of arcs during the next phase, then we should add the arc to the left-hand side, decide if the left-hand side can be divided into simpler subgraphs, and modify the right-hand side in a corresponding fashion.

3.2.2.2 Implementation Of The Interphase Model

Since the interphase model is merely a collection of graph rewriting rules describing the process of deriving one intraphase model from another, implementation of the interphase model should consist merely of recording the development process as it is carried out. To do this, we will require that the development team uses the discipline of recording the fact that a certain portion of a milestone document is to be replaced by a certain portion of a later milestone document. It is then necessary for a software tool to translate the

portions of the two documents into subgraphs of the interphase models of these two documents. This requires an ability to recognize the relationship between a software document and its model, but this recognition is achieved via the tools which implement the intraphase model. We have already demonstrated an implementation of this concept at the program code level, in which the text of the program and its internal representation are kept in step by means of a syntax-directed editor and an interactive prettyprinter. By analogy with that system, to record the fact that two subgraphs may be used to form one rule of the interphase model demands that we select the portions of the documents which correspond to each subgraph, then extract the portions of the internal representations which have been selected, and finally record the results as a part of the interphase model.

3.3 A Technique For Specifying Software Modification Proposals

In this section we will describe how to identify all items of a software system which may need to be changed as a result of a change request. At first we will describe how these items may be identified within the description of a particular phase ("intrapase tracing"), then we will describe how these items may be identified within the description of other phases ("interphase tracing"). A software modification proposal

consists of a list of all items which need to be changed, and a description, prepared by the maintenance programmer, of the change which must be made to each item.

3.3.1 Intraphase Tracing

The model of the system at each phase describes the control flow, data flow and data structures of the system during that phase. We are interested in tracing the effects of changes made to this phase of the system on other portions of the system. The problems are largely identical to those which we have already worked on for program modification, whose solutions we have called "logical ripple effect analysis" [YAU80b] and "performance ripple effect analysis" [YAU80c, 80f]. For that reason, we may use substantially the same approach for tracing the effects of changes during other phases. In fact, the problems of performing ripple effect analysis at the program level are reduced during other phases since the complex problems of aliasing and recursion are less likely to arise. In addition, the likely reduction in the size of the model to be traced makes ripple effect analysis techniques even more attractive. In developing our measure for design stability, we have already discussed the use of ripple effect analysis techniques at the design level. Hence, it will not be difficult to use that technique to detect potential

ripple effects for that phase. In addition, since the intraphase model has a similar structure, independent of any particular phase, the approach will also be independent of the phase at which it is applied.

In all of our previous work on analyzing the effects of program modifications, it has always been the case that we have restricted our definition of logical ripple effects to those potential changes in program behavior which may result from a change in the values of data items in the program. We have been able to identify which values may change by performing logical ripple effect analysis. Our later work on realizing program modifications helped us to identify another different, though relatively minor, type of ripple effect - the effects on the syntactic correctness of the program being modified. While those effects would be detected by a compiler, the ability of our program editor to detect them at the time the modification is being made is of great value to the maintenance programmer. For example, the ripple effects of a modification may lead to undeclared identifiers, because their declaration has been deleted. This kind of effects is to be handled by the syntax-directed editor which will be discussed in section 4.4.

This raises the question, then, whether there are other ripple effects of program modification which we are not yet able to detect. If so, how can we extend our approach to cover all of these effects? Furthermore, how can we be certain that no other types of modification effect can exist? Our response to these questions has been the development of a semantic model - with the intention of modelling all of the semantic properties of the system. By comparing our semantic model for a particular notation with the standard semantic definition for that notation, we can at least determine the completeness of our model for that notation. Thus, if no semantic changes must be made to our model of a system in this notation, we can deduce that no semantic changes will occur in the system. Since the "semantics" of a system is synonymous with its "logical behavior", it follows that no other logical ripple effects may occur. It is not so clear however, that no other performance ripple effects may occur. In addition, the proof of completeness must be carried out independently for each notation under consideration, and the notion of "completeness" must be understood to be limited by the completeness of the standard semantic definition of the notation (for example, certain decisions may be left to the implementors of the notation).

However, the fact that intraphase tracing is being performed in a multi-phase context makes it more probable that the results of tracing within a phase are reflections of similar tracing results at a previous phase. Therefore, these effects should be anticipated. Other results of the tracing phase will not have been anticipated; these are the effects due to the approach used in the implementation of this level. While it is clear that certain effects at one phase follow inevitably from changing a previous phase, these other effects appear to be less desirable - since they are not a consequence of the problem, but a consequence of the development process. These effects reduce the maintainability of the system as a whole and require the maintenance programmer to study more of the system before modifying it.

3.3.1.1 An Example Of RSL Modification

This example shows an RSL R_Net being modified. The R_Net is shown in Figure 3.15, and its MODEL representation is shown in Figure 3.16.

Now let us change ALPHA A1 to be

```
ALPHA: A1.  
  INPUTS: DATA: D1  
          DATA: D4.  
  OUTPUTS: DATA: D2  
           DATA: D3  
           DATA: D4.
```

```
R_NET: RN001.  
STRUCTURE:  
  INPUT_INTERFACE: I1  
  ALPHA: A1  
  DO ALPHA: A2  
  AND ALPHA: A3  
  ALPHA: A4  
  
  END  
  ALPHA: A5  
  OUTPUT_INTERFACE: O1  
END.
```

```
ALPHA: A1.  
  INPUTS: DATA: D1.  
  OUTPUTS: DATA: D2  
           DATA: D3.  
  
ALPHA: A2.  
  INPUTS: DATA: D2.  
  OUTPUTS: DATA: D4.  
  
ALPHA: A3.  
  INPUTS: DATA: D3.  
  OUTPUTS: DATA: D4.  
  
ALPHA: A4.  
  INPUTS: DATA: D4.  
  OUTPUTS: DATA: D4.  
  
ALPHA: A5.  
  INPUTS: DATA: D4.  
  OUTPUTS: DATA: D5.
```

Figure 3.15. RSL R_Net and associated alphas.

Then intraphase analysis should implicate activity A1 and data items {D2, D3, D4}. It may also be necessary to implicate the activities which provide the value of D4. These should then go on to implicate additional elements which use these implicated elements, as given in the data flow triples.

(MODEL)

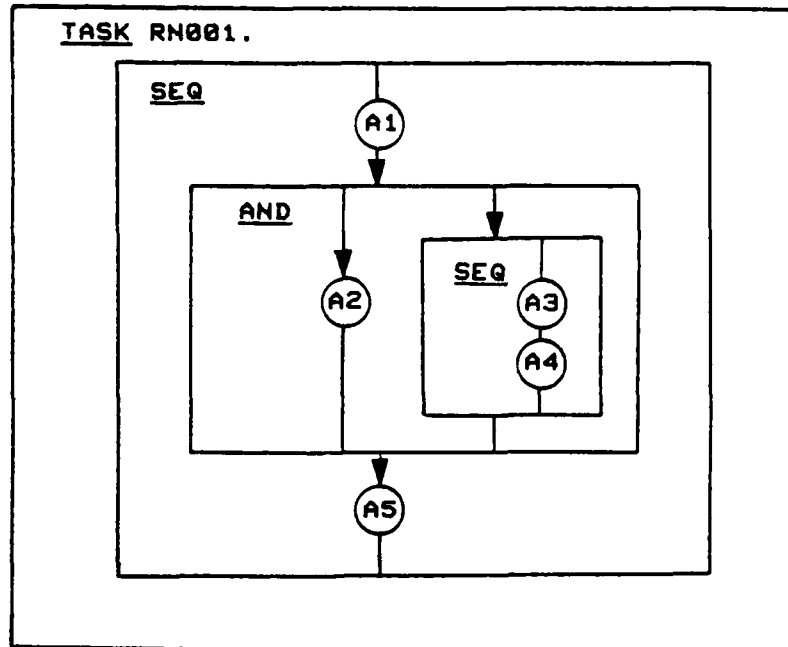


Figure 3.16. The MODEL representation of control flow of the RSL example.

3.3.1.2 Assertions To Control Intraphase Tracing

We may indicate assertions to the programmer which have been implicated, and let the programmer decide if any effects can propagate. These assertions should automatically be derived from the previous phase, and hence the effects on assertions may (perhaps) be derived or deduced from the effects observed at the earlier level. Of course, some assertions may

relate more to the implementation approach than to the problem area.

Now, since modifications are being made to the system, the assertions - since they state what the system state should be at a certain point in its execution - must also be changed. Hence we have the chart of possibilities shown in Figure 3.17. When the assertion is initially placed in the system, we will generally assume that it is correct - that it is easier to state the assertion than to write the program (segment). When the program is being modified, it is essential that all assertions be reexamined, since any "wrong" assertions will destroy all attempts to draw any conclusions about the program (see Figure 3.17).

		Code	
		OK	Wrong
A S S E R T I O N	OK	Pass	Fail
	Wrong	Pass/Fail	Pass/Fail

Figure 3.17. Consequences of possible combinations of good and bad code with good and bad assertions.

Even here, some interpretation of the terms "OK" and "Wrong" is necessary. Thus, an assertion may be passed by some segment of code, and the assertion may be guaranteed by the code, although the assertion which was needed to ensure the correctness of the program should have been stronger, and is not guaranteed by the code. We would have to consider such an assertion to be wrong because it does not accurately state what was required by that segment of code.

3.3.2 Interphase Tracing

We first illustrate our approach to interphase tracing with an abstract example, we then go on to define the procedure to be used for each type of "primitive" modification activity, and finally illustrate this with an example.

3.3.2.1 An Example Of Interphase Tracing

We will show how the effects of changes made to the software system shown in Figure 3.18 using the tracing rules of Figure 3.19 can be traced to the next level of system decomposition, shown in Figure 3.20. Note that, in the set of rules given in Figure 3.19, each node of the system appears on the left-hand side of exactly one rule. We follow this set of rules throughout our approach, since other rules involve more complexity both for design and tracing of the software system.

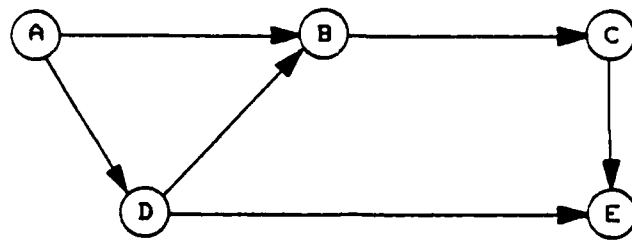


Figure 3.18. The abstract graph representing a software system.

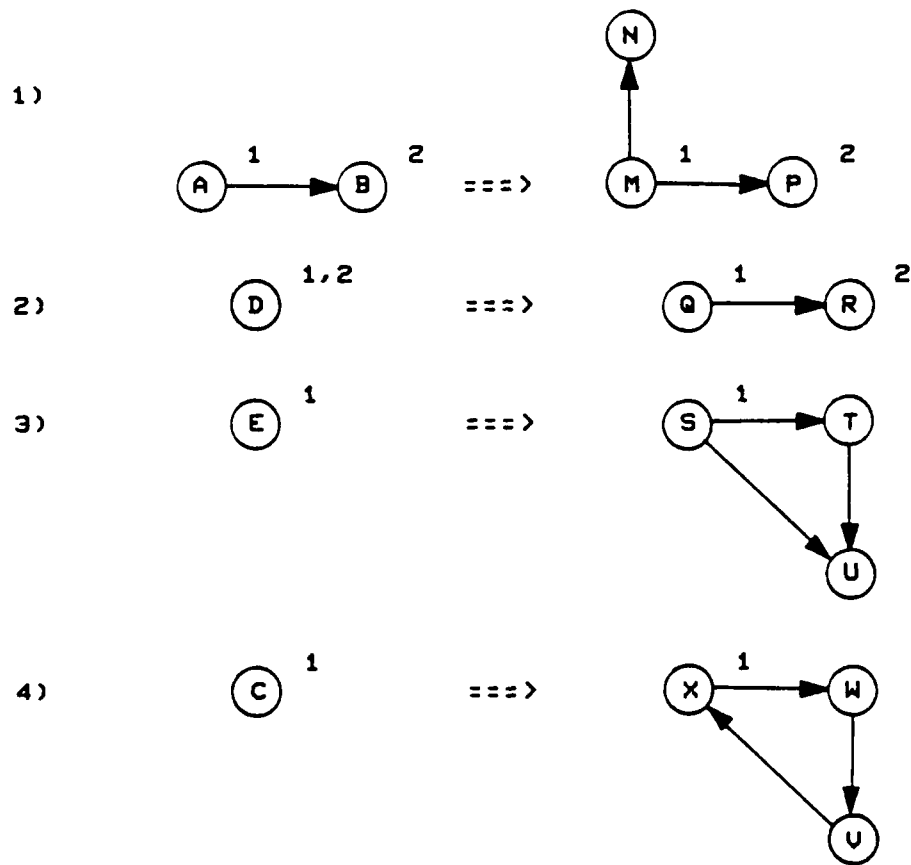


Figure 3.19. Tracing rules between two phases of an abstract software system.

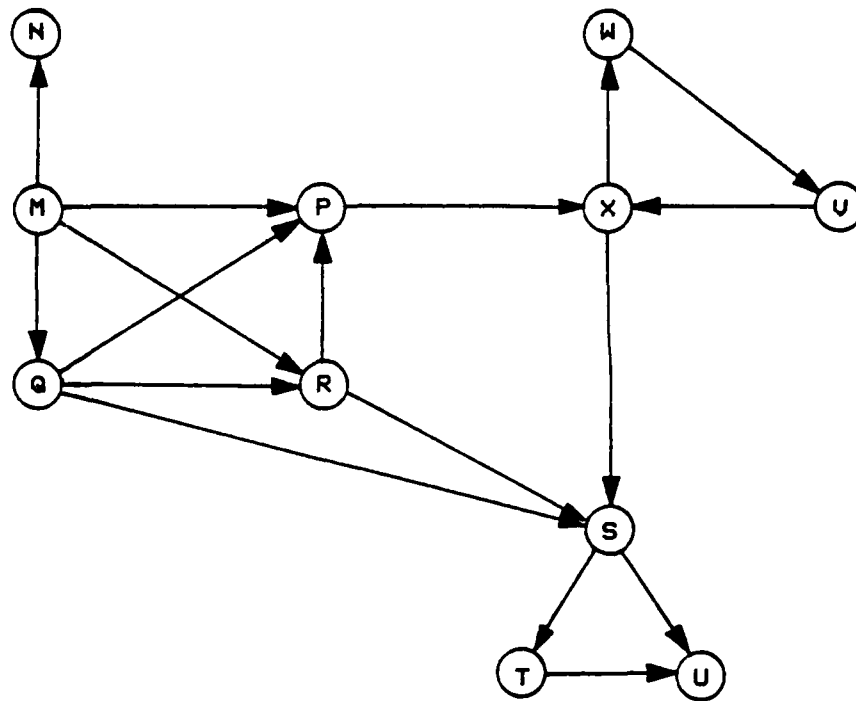


Figure 3.20. The next phase of the abstract software system.

Now, we would like to consider the effects of modifications to the original structure of the system.

3.3.2.1.1 Simple Modification

Consider to modify the node labelled C shown in Figure 3.18. This node seems to be fairly localized in the original graph and hence its ripple effect should not be too large. Let us show the detailed steps.

1. Local ripple effect analysis would require us to examine the nodes B and E (primarily E) because they are directly connected to node C. Let us assume that B and E need not be changed.
2. The tracing rules in Figure 3.19 show that the node labelled C appears in R4, being traced to a subgraph on the right-hand side of rule R4.
3. Now, we must determine what parts of the right-hand side of R4 must be changed. Let us assume that we decide to replace it by the alternate right-hand side of R4, which is shown in Figure 3.21, which we may consider to be the addition of a new feature (Y) together with the modification of an existing feature (U). Now, we must do some local ripple effect analysis of this new right-hand side of R4, as a result of the insertion of the node labelled Y and the modification of the node labelled U to U'. However, we anticipate that the right-hand side of a rule should be small enough to do the analysis thoroughly, perhaps even by hand.
4. Now, no further ripple effect analysis of the modified rewritten software system is needed, since the node labelled X is the only embedding item in the rule. Hence, if local ripple effect analysis has been done on node X (in

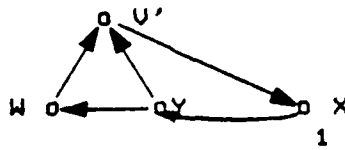


Figure 3.21. A new right-hand side for rule R4.

Step 3, the last step), no further ripple effects can occur.

5. Trace the effects to the next level.

- The modification to the node labelled U can be handled in this same way like the modification to the node labelled C at the previous level.

- The insertion of the node labelled Y must be handled differently, since one of the reasons is that there is no rule established for the new node (Y).

3.3.2.1.2 Insertions

This procedure should be followed for newly inserted nodes.

1. For each neighbor of the new node, determine if its rule (as before, we are assuming that there is only one rule for each node) should include the new node. If the node should be included, modify the left-hand side of the rule, and proceed as before for modifications to nodes (as for B and

U before). Otherwise, go to Step 2.

2. Define a new tracing rule, whose left-hand side is the new node, and whose right-hand side is a refinement of the semantic definition of the new node.
3. Determine how the right-hand side should be fitted into the new level. (The embedding problem).
4. Perform ripple effect analysis at the new level, to make sure that the new right-hand side "fits". Make new modifications as required.
5. Repeat the process for inserted and modified nodes at the next level.

3.3.2.1.3 Deletions

This procedure should be followed for nodes which are to be deleted. We will use the same example, with a new substitution for R4 to illustrate this procedure.

1. Let us assume that the alternate right-hand side of R4 is instead shown in Figure 3.22. Again, we should do some local ripple effect analysis of this graph. In this case, there is not much to inspect.

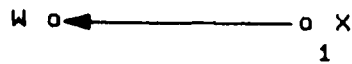


Figure 3.22. An alternative right-hand side for rule R4.

2. In addition (as before), since the node labelled X, the gluing item, is not affected, no further ripple effect analysis can occur.
3. Trace the effects to the next level.

Now, at the next level, we must first deal with the tracing rule involving the deleted node, labelled U. If the rule is a node replacement rule, then that rule may be deleted. Alternatively, since that rule will no longer be applicable, it may be left alone -- for "garbage collection". At this point, our strategy depends on whether we store the descriptions statically or store the tracing rules and allow the descriptions to be generated dynamically (the standard space/time tradeoff).

3.4 Discussion And Future Work

The results presented here deal with problems which have been largely ignored in the area of software engineering in general, and the area of software maintenance in particular. Yet, it is clear that we are dealing with problems which must

be resolved by software maintenance personnel if they would improve their productivity. This work is most closely related to that of software configuration management, in which the ability to trace software elements between different phases is emphasized, with the aim of improving the quality of a delivered software product. Our results represent a considerable improvement over software configuration management approaches, since we trace not only software elements, but also their interrelationships. This is particularly valuable to software maintenance personnel, who must eliminate all undesired side-effects of their modification activity.

The value of these results is limited by the absence of practical experience in using the approach with any software system. The effort needed to implement tools to support this approach would undoubtedly be considerable, even if these tools were restricted to particular well-defined notations for requirements, design and coding. In addition, we have not dealt with the question of the adequacy of representing only the control flow, data flow and data structures of a software system. Our model is a semantic model for software systems, most closely related to operational semantic definitions. Existing operational approaches have been used for software requirements [ZAVE81, 82], design [HAY74] and programming language definition [LEE72], [PAG81]. Hence, the success

achieved in these areas suggests that our approach is sufficient.

While some analysis of the approach remains to be performed, new questions have been raised by the results already obtained. First of all, it is clear that the approach has implications for the software development process. Currently, it is not customary for developers to record any information regarding the process of refining a system between different phases, although this information is clearly available. Using our interphase model this refinement process can be recorded, so that the maintenance personnel can make use of it. This leads us to ask if this information can be automatically extracted using other software tools in a software engineering environment. In addition, the realization that the refinement process will become a part of the system documentation should encourage software developers to consider how this process should be carried out. It is clear that the refinement process affects the quality of the final software system. Bowles [BOWL83] has shown that the complexity of a software design may be used to predict the complexity of a program developed from that design, under certain assumptions about the refinement process. His results may be considered together with our work to study the effects of different processes and the degree to which they permit additional

complexity to be introduced. Further studies might consider the effects on stability [YAU80e, 82c].

4.0 REALIZATION OF SOFTWARE MAINTENANCE PROPOSALS

Realizing a program modification proposal can be an expensive and unreliable process. We have developed an approach to program modifications more quickly and more accurately. Our approach uses a syntax-directed editor which operates on a formal model of the program. Using this editor ensures that modifications will always leave the program in a syntactically correct state. If a modification results in a syntactic inconsistency, this editor will advise the programmer of that fact and indicates where further modifications would be needed.

As an aid to the maintenance programmer, our approach will also use a program slicer [WEISB1, 82] in conjunction with the program editor to display those sections of the program which may affect the program code under investigation.

4.1 Overview

The overall procedure for our approach to this incremental process of program modification is shown in Figure 4.1. We assume that the programmer has made a preliminary decision as what types of changes must be made, based on a given modification request. Examples of the type of information which the programmer should have are the particular functions

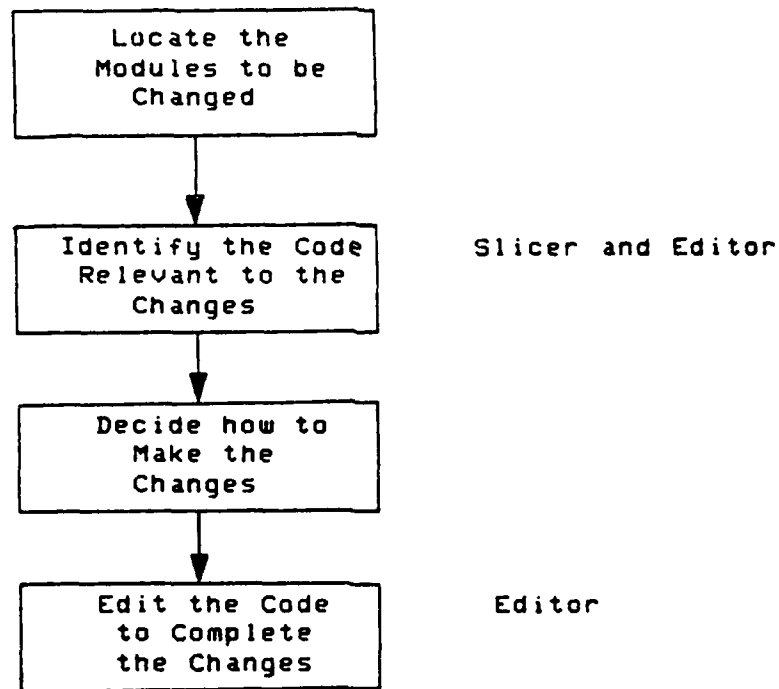


Figure 4.1. The procedure for incremental program modification

to be changed, the data values which are in error, or the additional functions required of the program. Our approach for the programmer to make the modification can be summarized as follows:

- (1) Based on the information obtained during the preliminary analysis of the proposed change, locate the program modules to which modifications must be made.
- (2) Use an interactive "program slicer" to identify the portion of the program which directly affects the program code and data values to be changed.
- (3) Decide what changes must be made to the code selected by the program slicer.
- (4) Use a syntax-directed editor to make the modifications to the program code. This editor will guarantee that the changes preserve the syntactic correctness.

In order to support this approach, we have developed a system which incorporates two major software tools: the program slicer and the syntax-directed editor. Figure 4.2 shows the organization of this system. The editor consists of three basic modules: an interactive pretty-printer for displaying the status of the program being modified, an incremental analyzer for analyzing the legitimacy of the modifications being made to the program and for updating data flow information, and a recursive-descent parser for parsing user-supplied textual information. A small routine, the "manager", is created for supervising the control flow of the

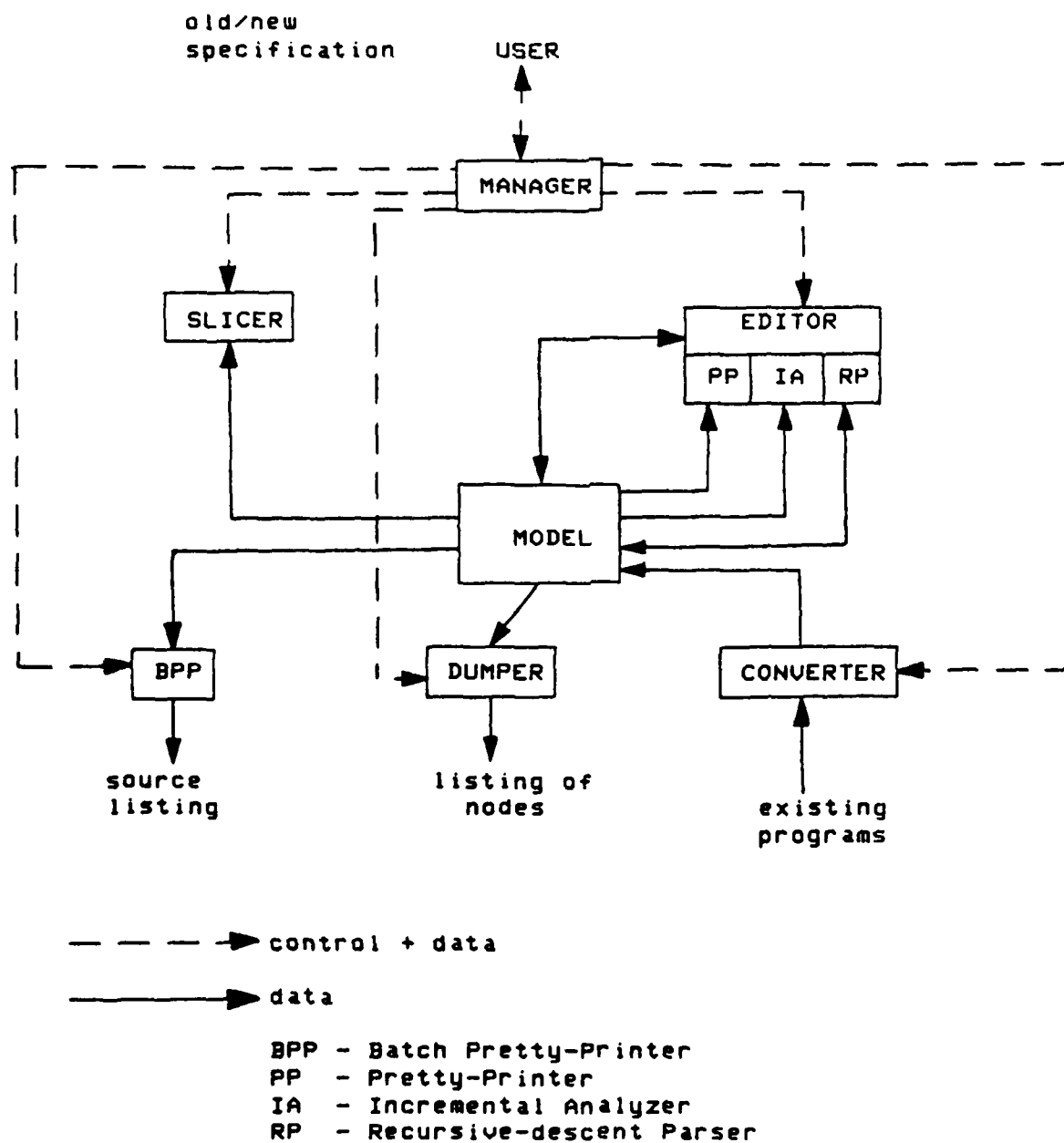


Figure 4.2. The structure of the system for incremental program modification

system. Upon receiving slicing commands from the user, the manager invokes the program slicer. Upon receiving editing commands from the user, the program editor will be invoked. A number of utility programs have been attached to the system: a converter for converting existing programs to our program representation (i.e. program model), a dumper for sequentially listing the nodes contained in the representation, and a batch pretty-printer for producing a well-indented source code listing.

4.2 The Program Representation

Most existing syntax-directed editing environments store the syntactic information of programs in the form of abstract syntax trees. Depending on the level of abstraction, there may exist a variety of abstract syntax trees. The hierarchical structure of a program is thus represented by the syntax tree. We feel that a program representation to be used in an interactive syntax-directed programming environment must meet the following criteria:

- 1) The representation must be formally defined, based on a formal specification.

- 2) The representation must be constructed without losing any of the syntactic information contained in the program.
- 3) The representation must present all features of the language in a uniform manner, so that a variety of tools can be easily integrated [WASS82].
- 4) The representation must support incremental program modification. That is, whenever a modification is made to a program, only part of the program needs to be updated and re-analyzed.

We have developed a tree-like representation for programs, which is based on the BNF notation frequently used for formally describing particular programming languages, and resembles the parse tree used by compilers. Our tree representation consists of a well-defined set of node types, each of which corresponds to a syntactic construct of the language. Definition of the representation for a particular programming language can be done using a procedure which operates on an annotated BNF description of the language.

4.2.1 Data Flow Extensions To The Basic Representation

In addition to recording the abstract syntax and static

semantics of a program, the program representation has been extended to include some data flow information. This data flow information takes the form of two attributes which list, respectively, the set of variables whose values may be used in that statement and the set of variables whose values may be defined by that statement. This data flow information can be constructed from the BNF notation for a simple PASCAL-like language as shown below, where the used variables are referred to by the attribute I (denoting input) and the defined variables are referred to by the attribute O (denoting output).

(1) $\langle \text{block} \rangle ::= \text{begin } \langle \text{statements} \rangle \text{ end}$

Here, a block can be the program main routine, a procedure or a function body.

$\langle \text{block} \rangle . I = \langle \text{statements} \rangle . I - \{ x | x \text{ is a local variable} \}$
 $\langle \text{block} \rangle . O = \langle \text{statements} \rangle . O - \{ x | x \text{ is a local variable} \}$

(2) $\langle \text{statements} \rangle ::= \langle \text{statement} \rangle$

$\langle \text{statements} \rangle . I = \langle \text{statement} \rangle . I$
 $\langle \text{statements} \rangle . O = \langle \text{statement} \rangle . O$

(3) $\langle \text{statements} \rangle' ::= \langle \text{statements} \rangle' ; \langle \text{statement} \rangle$

$\langle \text{statements} \rangle' . I = \langle \text{statements} \rangle' . I + \langle \text{statement} \rangle . I$
 $\langle \text{statements} \rangle' . O = \langle \text{statements} \rangle' . O + \langle \text{statement} \rangle . O$

(4) $\langle \text{statement} \rangle ::= \langle \text{assignment} \rangle ; \langle \text{procedure statement} \rangle ;$
 $\langle \text{for statement} \rangle ; \langle \text{while statement} \rangle ; \langle \text{repeat statement} \rangle ;$
 $\langle \text{if statement} \rangle ; \langle \text{case statement} \rangle ; \langle \text{compound statement} \rangle$

In this case, all the attributes are preserved.

(5) $\langle \text{assignment} \rangle ::= \text{id} := \langle \text{expression} \rangle$

$\langle \text{assignment} \rangle.I = \langle \text{expression} \rangle.I$
 $\langle \text{assignment} \rangle.O = \langle \text{expression} \rangle.O + \{ \text{id} \}$

(6) $\langle \text{procedure statement} \rangle ::= \text{id} (\langle \text{actual parameters} \rangle)$

In this case, let $\langle \text{block} \rangle$ be the corresponding procedure body, then

$\langle \text{procedure statement} \rangle.I =$
 $\{ x : \text{for each element } y \text{ in } \langle \text{block} \rangle.I, \text{ if } y \text{ is a formal}$
 $\text{parameter, then } x \text{ is used in the corresponding actual}$
 $\text{parameter, otherwise, } x = y, \text{ a global variable} \}$
 $\langle \text{procedure statement} \rangle.O =$
 $\{ x : \text{for each element } y \text{ in } \langle \text{block} \rangle.O, \text{ if } y \text{ is a formal}$
 $\text{parameter, then } x \text{ is the corresponding actual parameter,}$
 $\text{otherwise, } x = y, \text{ a global variable} \}$

(7) $\langle \text{for statement} \rangle ::= \text{for id} := \langle \text{expression}' \rangle \text{ (to; downto)}$
 $\langle \text{expression}'' \rangle \text{ do } \langle \text{statement} \rangle$

$\langle \text{for statement} \rangle.I = \langle \text{expression}' \rangle.I + \langle \text{expression}'' \rangle.I +$
 $\langle \text{statement} \rangle.I$
 $\langle \text{for statement} \rangle.O = \langle \text{expression}' \rangle.O + \langle \text{expression}'' \rangle.O +$
 $\langle \text{statement} \rangle.O$

(8) $\langle \text{while statement} \rangle ::= \text{while } \langle \text{expression} \rangle \text{ do } \langle \text{statement} \rangle$

$\langle \text{while statement} \rangle.I = \langle \text{expression} \rangle.I + \langle \text{statement} \rangle.I$
 $\langle \text{while statement} \rangle.O = \langle \text{expression} \rangle.O + \langle \text{statement} \rangle.O$

(9) $\langle \text{repeat statement} \rangle ::= \text{repeat } \langle \text{statements} \rangle \text{ until}$
 $\langle \text{expression} \rangle$

$\langle \text{repeat statement} \rangle.I = \langle \text{statements} \rangle.I + \langle \text{expression} \rangle.I$
 $\langle \text{repeat statement} \rangle.O = \langle \text{statements} \rangle.O + \langle \text{expression} \rangle.O$

(10) $\langle \text{if statement} \rangle ::= \text{if } \langle \text{expression} \rangle \text{ then } \langle \text{statement}' \rangle [$
 $\text{else } \langle \text{statement}'' \rangle]$

$\langle \text{if statement} \rangle.I = \langle \text{expression} \rangle.I + \langle \text{statement}' \rangle.I + [$
 $\langle \text{statement}'' \rangle.I]$
 $\langle \text{if statement} \rangle.O = \langle \text{expression} \rangle.O + \langle \text{statement}' \rangle.O + [$
 $\langle \text{statement}'' \rangle.O]$

- (11) $\langle \text{case statement} \rangle ::= \text{case } \langle \text{expression} \rangle \text{ of } \langle \text{cases} \rangle \text{ end}$
 $\langle \text{case statement} \rangle.I = \langle \text{expression} \rangle.I + \langle \text{cases} \rangle.I$
 $\langle \text{case statement} \rangle.O = \langle \text{expression} \rangle.O + \langle \text{cases} \rangle.O$
- (12) $\langle \text{cases} \rangle' ::= \langle \text{one case} \rangle [; \langle \text{cases} \rangle'']$
 $\langle \text{cases} \rangle'.I = \langle \text{one case} \rangle.I [+ \langle \text{cases} \rangle''.I]$
 $\langle \text{cases} \rangle'.O = \langle \text{one case} \rangle.O [+ \langle \text{cases} \rangle''.O]$
- (13) $\langle \text{one case} \rangle ::= \langle \text{constants} \rangle : \langle \text{statement} \rangle$
 $\langle \text{one case} \rangle.I = \langle \text{statement} \rangle.I$
 $\langle \text{one case} \rangle.O = \langle \text{statement} \rangle.O$
- (14) $\langle \text{compound statement} \rangle ::= \text{begin } \langle \text{statements} \rangle \text{ end}$
 $\langle \text{compound statement} \rangle.I = \langle \text{statements} \rangle.I$
 $\langle \text{compound statement} \rangle.O = \langle \text{statements} \rangle.O$
- (15) For $\langle \text{expression} \rangle$, if it does not involve any function call, then $\langle \text{expression} \rangle.I$ will be the set of variables used in the expression and $\langle \text{expression} \rangle.O$ will be empty. If a function call is involved, then the equations for $\langle \text{procedure statement} \rangle$ can be used to derive data flow information for that function call. The resulting data flow information will be the union of these two parts.

The minimum requirement for data flow analysis is that these attributes are attached to the nodes denoting conditional expressions or assignment statements. However, the representation described here is able to greatly reduce tree traversal, since we can immediately determine if a structured statement contains any references to a particular variable.

As a result of this extension to the basic program representation, a corresponding extension has been made to the editor's incremental analysis procedure, in order to keep these data flow attributes up-to-date while the program is being modified. Although we have only used these data flow attributes for performing program slicing, they can also be used for data flow analysis of a more general kind.

4.2.2 The Construction Of The Representation

For existing programs, a compiler-like process needs to be initiated to form the representation by generating a tree node for each language construct as soon as it is recognized by the parser. This process should present no problem, since existing programs in their production version are presumably both syntactically and semantically correct. The compiler or the interpreter of a particular programming language can be modified for this type of conversion. This conversion is, however, a one-time batch process. After the conversion has been carried out, the program representation is subject to modification, but this can then be handled by a syntax-directed editor. The editor is suitable not only for introducing new code into existing programs, but also for developing new programs.

4.3 The Program Slicer

4.3.1 The Concept Of Program Slicing

The purpose of "slicing" a program is to automatically extract sections of the program which are closely related to each other, with the aim of providing the information on which the programmer wishes to concentrate by removing those sections of the program which are not considered relevant to the modification task.

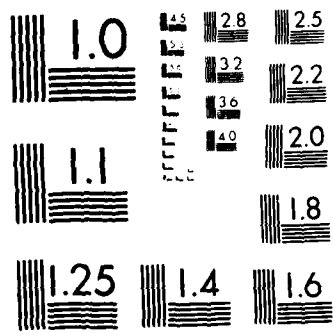
The term "program slicing" was first introduced by Weiser [WEIS81]. The interrelationships of program sections in a program slice were restricted to those which can be detected by data flow analysis. We also follow this restricted definition. A program slice can be constructed as follows:

- 1) Locate the statement in the program at which program slicing should start.
- 2) Decide which variables are of interest to the programmer.
- 3) Use data flow analysis techniques to identify all of the program which may affect the values of the selected variables.

Thus, the input to the program slicer consists of a program, a distinguished program statement, and a set of program

variables. The output consists of a set of statements of the program. The program slicer itself depends on data flow analysis techniques. The behavior of the statements selected by the program slicer will be partially equivalent to the behavior of the original program with respect to the selected variables and initial statement. Under the assumption that no non-terminating loop exists in the program, the behavior of the program slice and that of the original program with respect to the selected variables and initial statement are totally equivalent [WEIS81].

Weiser [WEIS81, 82] has shown that slices constructed in this way were recognized by subjects who, under experimental conditions, were asked to perform modifications to several programs. This result indicates that the subjects had (mentally) constructed program slices relevant to the modifications in order to modify the programs. However, the program slicer was not available for use by the subjects of the experiment. Furthermore, this program slicer operated on a conventional form of data flow graph [HECH77] (i.e. a directed graph whose nodes represent the condition and assignment statements of the program and whose edges represent possible control flow paths between them). Such a program slicer produces program slices with incomplete syntactic information to display a slice as a syntactically correct program.



MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS 1963-A

In an interactive programming environment, the slicer must present the programmer with a view of the program which corresponds to that presented by other tools in the programming environment. In this case, and in normal practice, this means that the text of the code in the slice must be displayed. Although the program slice has been defined in terms of data flow analysis and the selection of statements, in many programming languages, data declarations play a very important role. In such programs, it is necessary that program slices also include those declarations which declare all the objects used in those slices. Our program slicer meets these requirements, interactively constructing the text of partial programs which are made up of the subsets of declarations and statements of the original program which satisfy the slicing criterion and form a legal program. To achieve this, we extended a program representation, which we had developed to describe both the syntax and semantics of programs, to include the data flow information needed by the program slicer. Figure 4.3 illustrates the program slicing technique when applied to a small program.

Our current approach is based on an intramodule program slicer, which selects that portion of a module (i.e. procedure or function) which satisfies the slicing criterion, and includes declarations of objects inside or outside the module

```

type      . . .
  applerecord =
    record appletype: (golden, smith);
      rotten: boolean; order: integer;
      cost: dollars
    end ;
var      . . .
  i, count: 1 .. 20; accum: integer;
  average: real;
  apple: array [1 .. 20] of applerecord;

  . . .
  count := 0; . . .
  for i := 1 to 20 do
    with apple[i] do
      if not rotten then
        begin count := count + 1;
          accum := accum + order
        end ;
      if count > 0 then average := accum / count ;
    . . .

```

(a)

```

type      . . .
  applerecord =
    record rotten: boolean; order: integer;
    end ;
var      . . .
  i, count: 1 .. 20;
  apple: array [1 .. 20] of applerecord;

  . . .
  count := 0; . . .
  for i := 1 to 20 do
    with apple[i] do
      if not rotten then count := count + 1;
    . . .

```

(b)

Figure 4.3. (a) Portions of the program to be modified
 (b) Portions of the slice constructed for the variable COUNT.

which are necessary to ensure that the slice is indeed a syntactically correct program. In PASCAL [JENS74] these objects include labels, constants, types, variables, procedures and functions.

4.3.2 Algorithms For Syntax-Directed Program Slicing

To perform "syntax-directed" program slicing, we have developed the following algorithm, which operates on the parse tree of the program, to attach data flow sets to each statement and expression node of the tree.

The inputs to the program slicer are the augmented parse tree of the program, the point at which slicing should start (specified by the current position of the "cursor" within the parse tree) and a set of variables to be used to construct the slice.

The behavior of the algorithm depends on the particular statement type, based on the possible data flow paths which are permitted by the semantic definition of the statement type. The basic "generic" statement types are sequence, selection and iteration. In PASCAL-S, these are represented respectively by the compound statement type, the if and case statement types, and the while, repeat and for statement types. Other statement types may be collectively referred to as assignment statements.

The basis of the slicing algorithm may be written as follows:

```
procedure slice (St: statement ;  
                 var SU: set of variable names) ;
```

```
comment
```

This procedure identifies the statement type of St, and uses the value of SU to determine if any statements within the parse tree rooted at St should be included in the slice. The value of SU is updated to reflect the effects on the possible data flow of the behavior of St.

```
end comment
```

```
if  $SU \cap St.output \neq \emptyset$  then  
  case St.statement type of  
    SEQUENCE: slice_sequence (St, SU) ;  
    SELECTION: slice_selection (St, SU) ;  
    ITERATION: slice_iteration (St, SU) ;  
    ASSIGNMENT: slice_assignment (St, SU) ;  
  end case  
  include_statement (St)  
end if ;
```

```
end procedure
```

where the sub-procedures are defined as follows:

```
1) slice_sequence (St: statement ;  
                  var SU: set of variable names) ;  
   slice (youngest_unsliced_child_of (St), SU)  
   while St has more unsliced children do  
     slice (youngest_unsliced_child_of (St), SU)  
   end while
```



```

2) slice_selection (St: statement ;
                   var SU: set of variable names) ;
   OSU := SU ; T := OSU ;
   slice (last_unsliced_choice_of (St), T) ;
   NSU := T ;
   while St has more unsliced choices do
     T := OSU ;
     slice (youngest_unsliced_choice_of (St), SU) ;
     NSU := T U NSU
   end while ;
   SU := NSU U Expr.Inputs ;
   include_expression (Expr)

3) slice_iteration (St: statement ;
                   var SU: set of variable names) ;
   OSU := T ; T := OSU ;
   slice (body_of (St), SU) ;
   NSU := T ; T := T - OSU U Expr.Inputs
   while T  $\neq$   $\emptyset$  do
     OSU := OSU U T ;
     slice (body_of (St) SU) ;
     NSU := NSU U T ; T := T - OSU
   end while ;
   SU := NSU ;
   include_expression (Expr)

4) slice_assignment (St: statement ;
                    var SU: set of variable names) ;
   SU := SU - St.Outputs ;
   SU := SU U St.Inputs ;

```

The slicing algorithm progresses by traversing the parse tree in an order which visits the statement nodes which precede the initially chosen statement node (according to the program's control flow) in reverse control flow order. When structured statements are encountered, they are considered in a top-down order, being considered only while their output data set (the variables affected by that statement) overlaps with the current set of "slice variables".

All statements chosen by the slicing algorithm are "included" in the resultant slice of the program. To ensure the correctness of the syntax of the slice, declarations of any objects used in the included statement must also be included in the slice. These "objects" include named constants and variables, and their associated type definitions, as required.

In the case of our intramodule slicer, we must also devise an approach to deal with calls to other modules. When such calls are included, we have adopted the convention of including an empty version of the called procedure in the slice. This version of the procedure includes its name, type (if any) and formal parameter list, together with an empty declaration part and an empty compound statement for its body. This is sufficient to satisfy the syntactic requirements of the language.

Since block structured languages such as PASCAL permit access to "objects" declared at any one of several upper levels, we have chosen to preserve the upper levels in the body of the slice. Thus, the slice will include "empty" versions of all procedures which contain the module being sliced. Within each of these procedures will also appear the declarations of any objects which were previously declared in that procedure, and which are needed within the slice. Clearly, the inclusion of declarations within the slice is important for helping

program modification.

The nodes which are included within a slice form a subset of the nodes in the parse tree of the program. However, they can be rejoined to form another parse tree, using the edges which existed in the original tree as a guide. The new parse tree constructed in this way is used to display the text of the slice identified by the slicing algorithm.

However, these algorithms are not sufficiently general to allow the programmer to select statements arbitrarily. For instance, if the programmer selects a statement from the center of a sequence of statements, the "slice_sequence" procedure must be altered to start slicing from the selected statement, instead of starting from the last statement in the statement sequence. To handle this and similar cases involving "slice_selection" and "slice_iteration", we have written modified algorithms to perform slicing on partial parse trees.

First of all, it is necessary to construct a list L of all statements which enclose the selected statement. This list can be constructed in a straightforward manner from the parse tree, by visiting the "parent" of each node until the body of the module is reached. The following algorithm, a modified form of the "slice" procedure, is used:

```

procedure part_slice (St: statement ;
                      L: list of statements ;
                      var SU: set of variable names) ;

if L is empty then slice (St, SU)
elsif SU  $\cap$  St.Output  $\neq \emptyset$  then
  case St.statement type of
    SEQUENCE: slice_part_sequence (St, L, SU) ;
    SELECTION: slice_part_selection (St, L, SU) ;
    ITERATION: slice_part_iteration (St, L, SU) ;
    ASSIGNMENT: slice_assignment (St, SU) ;
  end case
  include_statement (St)
end if ;

end procedure

```

As an illustration, we show the modified form of the procedure "slice_sequence". Similar modifications must be done to "slice_selection" and "slice_iteration".

```

slice_part_sequence (St: statement ;
                    L: list of statements ;
                    var SU: set of variable names) ;

part_slice (head (L), tail (L), SU)
while St has more unsliced children
  preceding head (L) do
    slice (youngest_unsliced_elder_sibling_of (head (L)), SU)
  end while

```

The initial call to start slicing will be:

```
part_slice (head (L), tail (L), SU).
```

4.3.3 Enhancements

To improve the usefulness of this program slicer as a programming aid, we have added the options of further applying the slicer to existing slices of a program to obtain a more refined picture of program behavior and of combining slices (possibly those of distinct modules) into more comprehensive units. We have defined the following operations for combining program slices into larger units, including statements taken from several modules:

UNION: Given two slices, S1 and S2, construct a third slice S3 which contains all the statements and declarations which appear in either S1 or S2.

INTERSECT: Given two slices, S1 and S2, construct a third slice S3 which contains all the statements and declarations which are common to both S1 and S2.

By the definition of a program slice, each of these operations will always ensure that the slice S3 satisfies the requirements of a program slice, and also ensures that it will be syntactically correct. Since a program slice can be considered as a parse tree, or even as a set of nodes taken from a parse tree, these operations are readily implemented using well-known algorithms for set operations.

4.4 The Syntax-directed Editor

We can make the following observations on existing syntax-directed editors: They are designed specifically for program development, emphasize the creation of programs in a top-down fashion, are based on the abstract parse tree, incorporate an incremental semantic evaluation mechanism, and are highly experimental in nature.

One major contribution made by existing syntax-directed editors is that a program is treated as a well-formed collection of syntactic units (language constructs), not just text. The actions carried out by these editors can be classified as syntactic editing operations because the syntactic structure of the program will be affected as an immediate result of these operations. The programmer using these "syntactic" editing operations should, however, expect "semantic" effects as well. Most program editors do perform semantic checking, which is enforced in conjunction with the syntactic editing operations.

In this section we briefly describe a new type of program editor which also supports incremental analysis and update using a tree representation of programs, and displays program text using a screen-oriented pretty-printer. The editor, however, is based on the class of editing operations which are

termed semantic editing operations, in the sense that not only the syntactic structure of the program is affected, but also each of these operations has a meaning (semantics) which is defined by the context in which the operation is performed.

For example, suppose that the cursor is positioned over a constant definition. The programmer can add a new constant definition, appearing after the current one, by issuing an insert operation. The programmer does not have to explicitly specify the intention to insert a new "constant" definition. Knowledge of the immediate semantic effects of the editing operation is therefore shared between the programmer and the system. More complicated semantic effects, such as multi-declarations, are still subject to tracing by the system alone.

There are at least two major advantages in using this kind of editing operations:

1. Since the programmer is made aware of the structures of the programming language, modifications are performed as operations on these structures, rather than as operations on a piece of text. We believe this to be a more reliable and informative way of modifying programs, although certain textual operations are still valuable.

2. Less information needs to be provided by the programmer because the cursor position helps the editor to determine the meaning of each operation.

We have defined three classes of commands, basic modification commands, cursor movement commands, and extended modification commands. Programmer's modifications can be translated in the underlying operations for each command. The programmer's view of the editing operations, however, uses a more friendly notation than the commands described in that paper.

4.4.1 Incremental Editing

Very often one may prefer, at intermediate stages of program editing, some syntactic structures of a program to be temporarily incomplete. Therefore, the concepts of "templates", "placeholders" and "phrases", as described in the Cornell system [TEIT81], are also used in our system. These concepts are illustrated in the following example:

insert a "while" statement after the "for" statement

```
.  
.  
  for i := 1 to 20 do apple[i] := pie[i];  
  while <<condition>> do <<statement>>;  
.  
.
```

"while" template
(construct)

placeholders
(components of construct)

"Phrases", which we call "primitive strings", are subject to parsing. A simple "recursive descent" parser is included in our system to perform this limited parsing. The process is incremental only in the sense that, after parsing, the resulting subtree is included in the existing program tree.

The set of basic modification commands is suitable for updating programs in a more incremental manner, while the set of extended modification commands takes advantage of the existing program constructs.

4.4.2 Legitimate Operations

Not all types of editing commands can be applied to each language construct. For example, in PASCAL, the DELETE operation can be applied to the "ELSE" part of the IF_THEN_ELSE construct to delete the keyword ELSE and all the statements of the "ELSE" body. The operation, however, may not be applied to the "THEN" part of the IF_THEN_ELSE construct. Note that all the statements of the "THEN" part can be deleted to leave an empty "THEN" part.

We have defined a Legitimate Operation Table which records, for each language construct, the type of semantic editing operations that can be applied. Figure 4.4 shows part of the table for the programming language PASCAL. Whenever the

programmer specifies an operation to be performed, the editor must consult the table to determine the legitimacy of the intended operation.

4.4.3 Incremental Analysis

The major function of incremental analysis is to perform incremental evaluation of the static semantics of programs. According to the characteristics of the operation, the current cursor position in the program representation and the new information to be included in the case of ADD, INSERTA, INSERTB and CHANGE operations, consistency checks of the static semantics of the program being modified must be made.

For each entry in the Legitimate Operation Table, certain semantic "hooks" may be defined. These semantic hooks trigger the invocation of related semantic checking routines, when the entry indicates that the operation is legitimate. For example, the command to change an assignment statement "a := b+c" to "a := b+d" may be hooked to three semantic checking routines:

1. Check whether the variable "d" has been declared or not.
2. Check whether the variable "d" can be used as an operand in this statement, according to its type.

Operations	ADD	INSERTA	INSERTB	CHANGE	DELETE
Language Constructs					
CONST	X	O	O	O	O
VAR	X	O	O	O	O
PROCEDURE_CALL	O	O	O	X	O
ACTUAL_PARM	X	O	O	O	O
BEGIN_END	O	O	O	X	O
IF_THEN_ELSE	O	O	O	X	O
THEN	O	O	X	X	X
ELSE	O	X	X	X	O
WHILE	O	O	O	X	O
EXPRESSION	X	X	X	O	X
TEMPLATE	X	X	X	O	X

O : legitimate operation
X : illegitimate operation

Figure 4.4. A part of a legitimate operation table.

3. Perform type coercion.

Since temporary semantic inconsistency at intermediate stages of the program modification activity should be tolerated, the language constructs involved may be highlighted to indicate the violation until it is removed. For example, if a variable declaration is deleted, a list of usages to this variable in the program will remain, in which each element represents a semantic inconsistency (i.e. an undeclared variable). The system should assist the programmer in identifying this list of semantic inconsistencies.

Figure 4.5 shows a more complex case, in which a new variable is introduced into a procedure B which is nested within another procedure A. The original variable CURSOR was declared in procedure A, and used in both of the procedures A and B. If a new variable CURSOR is declared in procedure B, this will override the previous declaration. A very likely consequence is that the usages in procedure B of the original variable CURSOR will also become invalid.

This may be because the attributes of the two variables are totally different. Even if these two variables have identical attributes, the programmer's intention is still

```
Procedure A;  
  Var  CURSOR : integer;  
  
  Procedure B;  
    . . . ←  
    begin (* procedure B *)  
      { CURSOR used }  
    end; (* procedure B *)  
  
  begin (* procedure A *)  
    { CURSOR used }  
  end; (* procedure A *)
```

Figure 4.5. Insertion of a local variable.

unknown. If these two variables have identical attributes, a compiler must take into account the new declaration, and apply it to all the "usages" of the variable CURSOR declared in procedure B. However, since the programmer was making modifications to an existing program, he might not be aware of the existence of another variable of the same name. Compilers are obviously ineffective in detecting this kind of ("injected") error.

By comparison, a highly responsive program editor can assist programmers in detecting them at the earliest possible stage. We do, therefore, feel that it is the responsibility of the editor to inform the programmer about such dangers, and to require the programmer to resolve the ambiguity.

4.4.4 Incremental Update Of Data Flow Information

Once a modification is made to a node of the tree model, the data flow equations are used to update the data flow information for that node. Since the data flow information for most nodes is derived from its children, changes will be propagated to the ancestors of the modified node as far as possible. If we let the propagation of changes proceed each time a modification is made, we will find that there are two immediate disadvantages. First, this propagation for large-scale software systems may continue for a long time, if the next modification is made to a descendant node of the current node, this propagation of changes is not only wasteful, but also unnecessary.

Since the cursor movement along the parse tree is continuous, we realize that updating data flow information for the current node should be done only when the next move is to a sibling node or to the parent node. This scheme reduces the response time significantly and still guarantees that data flow

information is ready whenever the subtree rooted at the current node is referenced. Whenever a slicing command is entered, propagation will be performed until the propagation reaches the root of the tree or stops at some node which has no change in its data flow information.

One of the major assumptions of the above scheme is that we assume that only the nodes which lie on the path from the root of the current block to the current node have incorrect data flow information. Otherwise, we assume that the data flow information of descendant nodes and sibling nodes is correct at any instant, even when procedure statements or function calls exist. This is not true if we do not update the data flow information of procedure statements and function calls when the data flow information of the corresponding block changes. We consider these changes as side effects which are created when the data flow information of a block is changed. In this case, all the procedure statements or function calls which refer to this block must also be updated, and we must propagate the change in data flow information as far as possible.

4.4.5 Interactive Pretty-printing

The function of the screen-oriented pretty-printer is to allow the programmer to view the portion of the program being edited. The programmer first uses the cursor commands to examine the program, then uses the editing commands to modify the program. The pretty-printer responds to cursor commands, and rebuilds the screen display according to program changes, by examining the program representation. As a result, the pretty-printer provides instant visual feedback to assist the programmer in perceiving program changes in an interactive manner. Figure 4.6 shows the various cursor positions resulting from a sequence of cursor movements.

4.5 Software Development

We are currently completing an implementation of a prototype version of the system shown in Figure 4.2. The system has been written in PASCAL and runs on our VAX-11/780 computer. Our choice for the first target programming language is PASCAL-S, a subset of PASCAL [WIRT75]. The program representation is implemented as a set of fixed length PASCAL records, each of which corresponds to a construct in the PASCAL-S language.


```

procedure sort;
*7
  var counter, pointer, temp : integer;
  begin
    counter := 20;
    while counter > 1 do
      begin
        pointer := 1;
        *1
        while pointer < counter do
          *2   *3
            begin
              *4
                if list[pointer] < list[pointer+1] then
                  *5
                    begin
                      temp := list[pointer];
                      list[pointer] := list[pointer+1];
                      list[pointer+1] := temp;
                    end;
                  pointer := pointer + 1;
                *6
              end;
            counter := counter - 1;
          end;
        end;
      end;

```

position	*1	to position	*2	:	DOWN
"	*2	"	*3	:	RIGHT
"	*3	"	*4	:	DOWN
"	*4	"	*5	:	RIGHT
"	*5	"	*6	:	DOWN
"	*6	"	*7	:	DIAGONAL

Figure 4.6. An example to show a sequence of cursor movement.

To convert existing PASCAL-S programs to the program representation, we have modified the PASCAL-S interpreter by Wirth [WIRT75] so that we can use the syntax-directed editor to modify the program. Of course, the editor can also be used for new program development. Our implementation of the pretty-printer has been enhanced by using an "extended cursor" [TEIT81] to highlight an entire programming language construct. The program slicer is now operational on individual modules. However, using the operations UNION and INTERSECTION of program slices it is possible to construct program slices using intermodule data flow.

These software tools (modules) communicate with one another through updating and examining the value of the current position indicator in the tree, given by a global variable TREE_CURSOR. Figure 4.7 shows the communication pattern. The utility programs described in Section 4.1 have also been implemented, and they are often used as off-line tools. To provide the programmer with more information about the program and the status of the modification, we use a multi-display system. Two CRT terminals are used simultaneously one for displaying program fragments and the other for user interface. This will allow functions such as issuing commands, entering character strings (primitive information) and receiving system

messages.

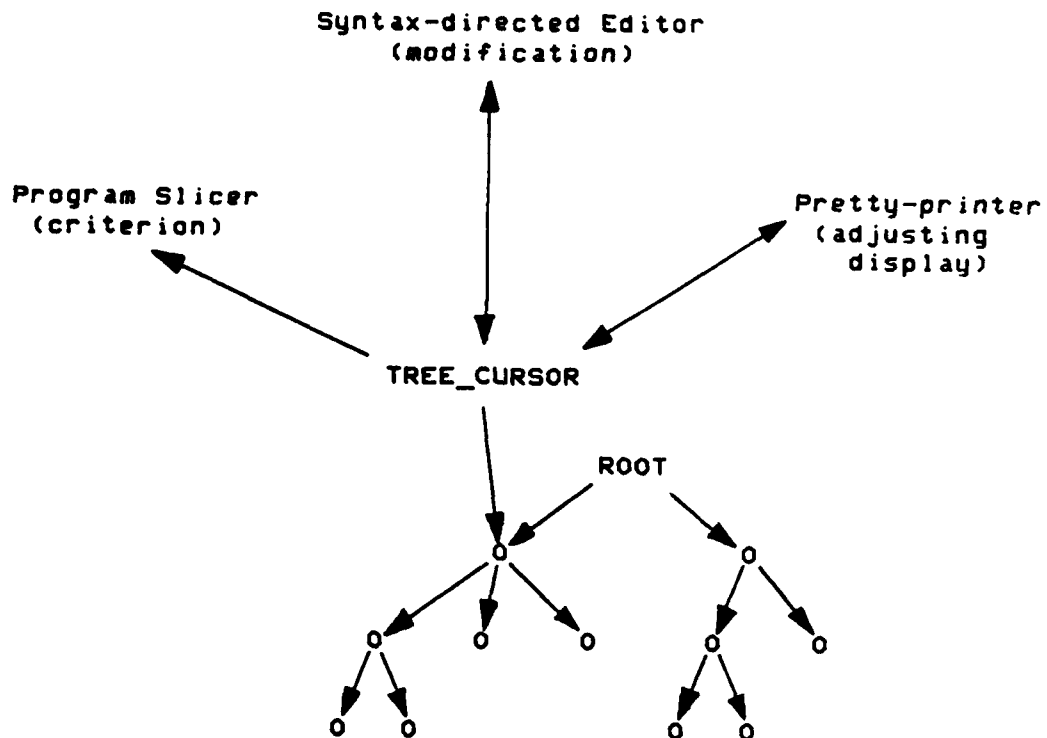


Figure 4.7. The communication pattern of the integrated tools.

By integrating these tools using our program representation, we have provided an environment in which different activities involved in program modification may be

coordinated and treated as parts of a single task. Our experience based on testing early versions of these modules (such as the program slicer and the pretty-printer) indicates that our approach is feasible. Consistently using the tree operations defined on the program model as editing operations has been shown to be practical.

4.6 Discussion And Future Work

We have presented an approach to incremental program modification using a set of well-integrated software tools. We have also presented a tree-like program representation which contains sufficient information about the program structure and static semantics with data flow extension to facilitate various analysis.

In order to use this approach, the following improvements need to be made:

1. Programmers are allowed to move freely to any spot in the program by means of the structured cursor movement commands. It is found that the correspondence between the position in the representation and the user's view of the position in the text is troublesome, and that different nodes within the tree representation often correspond to the same piece of text. This would confuse the user as to

the exact location of the cursor in the program. Our solution to this problem will be to refine our implementation so that movement commands automatically skip certain nodes which do not correspond to a distinct construct in the user's view of the text.

2. The extended cursor provides a visual cue for the programmer by clearly highlighting the current construct. Movement commands with even larger spans are still needed for the programmer's convenience.
3. The set of editing commands is complete in the sense that it allows any kind of modification. However, in order to achieve greater efficiency, this set must be extended. For example, multiple buffers can be introduced to facilitate more powerful refinement actions (such as combining two sections of code into a single construct).

From the previous discussion, it is clear that further research is needed to have a better environment for program modification. For instance, in order to use this approach to different programming languages, the construction of the program representation should be at least semi-automated. This should be feasible because the program representation and the operations on it are formally defined. Furthermore, since separate compilation is a very important and useful feature of

programming languages, a practical syntax-directed programming system should have facilities to support this feature. In addition, the program modification system should easily incorporate many other software tools, such as ripple effect analyzer, which will be discussed in the next section.

5.0 RIPPLE EFFECT ANALYSIS

One of the most serious problems facing the maintenance programmer is to accurately determine the consequences of making a particular program modification. While visual inspection can be successful, automated analysis techniques are likely to be more reliable. We have developed an approach to perform automated analysis of the ripple effects of program modification, and this approach has been demonstrated using PASCAL programs on a DEC VAX-11/780 computer. The analysis technique may be used to identify potential ripple effects on both the logical and performance aspects of program behavior. The logical ripple effect analysis technique is a significant improvement over that previously demonstrated for JOVIAL programs [YAU78, 80a, 80b] and is able to deal with the problems of recursion and dynamic aliasing. In this section, we will present both our logical and performance ripple effect analysis techniques.

5.1 Logical Ripple Effect Analysis Technique

The logical ripple effect analysis technique presented here is to statically analyze the changes to the data flow of the program introduced by an initial program modification. When the value or attribute of a variable in one portion of the program may be changed after an initial program modification,

the variable may cause potential errors when it is used. Thus, this variable is identified as a potential error source. As a simple example, consider the following program segment:

```
S1 : x := x + 1;  
:  
:  
S2 : y := x + z;
```

Suppose that the expression on the right-hand side of the first assignment statement S1 is modified in an initial program modification, then the assignment of y in S2 may become logically inconsistent with the initial modification.

Similarly, when a control condition, e.g. `if (x > y)`, is changed in an initial program modification, potential errors may be introduced to the program since the execution and hence the result of the program may be changed.

A potential error source can be a primary or a secondary error source. A primary error source is a variable or control condition whose value or attribute is modified by an initial program modification. A secondary error source is then a variable or control condition whose value or attribute may become inconsistent with the initial program modification. In the above example, x in S1 is called a primary error source, while y in S2 a secondary error source. The propagation of the potential error sources will be referred to as potential error flow.

To identify the potential error flow, our logical ripple effect analysis technique identifies and utilizes the definition and usage information commonly used in data flow analysis techniques [ALLE74], [LOME77], [BART78], [ROSE79], [ARTH81]. In the above example, our logical ripple effect analysis technique will identify y as a secondary error source based on the information that the definition of y in S2 uses x, which is a primary error source. Hence, the scope of logical ripple effect which can be identified using our technique is bounded by the capabilities of its underlying data flow analysis technique.

Our logical ripple effect analysis technique is similar to those program analysis tools, such as DAVE [FOSD76] and program slicing technique [WEIS81], in that they are all based on data flow analysis of the program. However, they differ in their applications of the data flow information. For example, DAVE is concerned with identifying the data flow anomalies of a program, while program slicing technique is focused on identifying an executable "slice" of a program which may result in the definition of a variable at one point of the program. Both DAVE and program slicing technique are not applicable in identifying the logical ripple effect of an initial program modification, because they do not identify the changes to the data flow of the program after an initial program modification.

Our technique, on the other hand, provides a trace of the program segments which may be affected by the logical ripple effect of an initial program modification.

In this section, only the framework of our logical ripple effect analysis technique is presented through the development of abstract models. These models can be applied on sequential programs written in high level languages such as FORTRAN, PASCAL, etc. Implementation or language specific details are not discussed here.

Our technique performs logical ripple effect analysis in two stages. The first stage is the error flow model construction stage, during which an intramodule error flow model and then an intermodule error flow model will be constructed to characterize how potential error sources can propagate within the modified version of the program. The intramodule error flow model characterizes how potential error sources can propagate within the modules in the program. The intermodule error flow model characterizes how potential error sources can propagate between the modules in the program. The construction of the intramodule error flow and the intermodule error flow models have approximately the same level of complexity as the intramodule and intermodule data flow analyses, respectively.

The second stage of our logical ripple effect analysis is the logical ripple identification stage which concerns with identifying the potential error sources implicated by an initial program modification. This stage can be performed in two phases. During the first phase, the primary error sources are identified based on the initial program modification and the error flow models of the modified program. Then, in the second phase, the logical ripple effect will be traced utilizing the primary error sources and the error flow models.

The logical ripple effect analysis technique presented here is capable of providing exhaustive tracing of the logical ripple effect. It can be tailored to support other strategies for logical ripple effect tracing. For instance, an implementation of the technique may provide only intramodule error flow tracing which can be sufficiently effective in an environment where intramodule error flow dominates, while the cost of applying this technique can be greatly reduced. Another example of an implementation of this technique is to identify only the error sources directly implicated by the primary error sources.

A prototype logical ripple effect analyzer for PASCAL programs has been developed. This analyzer provides an interactive environment for tracing the logical ripple effect. The extent of the logical ripple effect tracing can be

controlled by the software maintenance programmer such that he can choose the program areas of his interests to be examined by the logical ripple effect tracing scheme. Also, the software maintenance programmer can eliminate some modules or variables from the logical ripple effect tracing, which are not affected by the initial modification based on his understanding of the program. Thus, our logical ripple effect analysis technique can identify the program areas which will require additional maintenance effort. Some experimental results of our logical ripple effect analysis technique for PASCAL programs will also be presented.

5.1.1 Intramodule Error Flow Model

In this section, we will present the intramodule error flow model, and show how the propagation of potential error sources within the modules in a program can be modelled by the model. Before we present these, we need to make a number of definitions.

A program module is defined to be a separately invokable piece of code having a single entry and a single exit. Practically speaking, a module can correspond to a SUBROUTINE or PROCEDURE, etc. To reduce complexity, a program module is further represented as a set of program blocks. A program block can be either a local block or an external block. It

will be seen later that there is a sequence of three blocks in the invoking module for each module invocation, which can be a procedure call statement or a function reference; and these three blocks for each module invocation are called external program block. A local program block contains an expression which provides a control condition, or a simple statement other than a procedure call statement. Each program block has a single entry and a single exit. However, a program block may reach or be reached by several program blocks. For example, a program block containing an "if" clause may reach two program blocks corresponding to the "then" and "else" parts of the "if" statement.

The flow of control among the program blocks of a module can be represented by a control flow graph associated with this module. The control flow graph associated with a module m can be expressed as a quadruple, $CFG[m] = (U, B, u, v)$, where U is the set of vertices representing the set of program blocks in the module m , B is the set of branches which are ordered pairs of vertices representing the flow of control from the exit point of a program block to the entry point of another program block, u is an element of U representing the entry block of the module m , and v is an element of U representing the exit block of the module m . Note that the entry and exit blocks of m are used to trace the error flow into and out from m . They do not

correspond to any executable statements in m.

The intramodule error flow analysis can be simplified by decomposing the error flow within a module into the error flow which occurs within a program block and the error flow which occurs between program blocks in a module. In order to analyze the error flow within program blocks and between program blocks, it is necessary to develop a characterization for a program block which reflects how potential error sources may flow within the program block.

5.1.1.1 Block Error Characteristics

The basis for the characterization of a program block requires the identification of all data items and control items in the program block. A data item is a member of the set of minimal information units which describe the program. They basically consist of the program's variables. The control items are artificially created in our logical ripple effect analysis to provide a basis for linking the data flow and control flow information together in the program. A control item is created for each control condition which determines the execution of a statement or a group of statements. For example, the predicate in a conditional "if" statement provides a control condition which determines the outcome of this decision point, and hence a control item is created to

represent the predicate. A FORTRAN "do" statement which establishes a controlled loop also provides one type of control item. A control item can be created in such a manner that it will not generate any erroneous error flow in the program by assigning to it a symbolic name which is guaranteed to be distinct from any identifier in the program and from any other control item. A definition is an item whose value is modified or read in a part of a statement, or whose associated control condition is defined in an expression. A usage is an item whose value is referenced in a part of a statement, or whose associated control condition can affect the execution of the statement.

It is assumed in our error flow analysis that all data items have a unique memory address and that this memory address can be symbolically determined prior to program execution. This implies that the variables with the same name but different scopes are treated as different data items. It also implies that all the elements in a data structure are represented by the data structure itself. Due to the static nature of our analysis, it is infeasible to trace the exact error flow for programs which contain data structures. However, the worst-case error flow can be computed by treating a data structure as a single data item. Thus, if an element in a data structure is affected by the error flow, the whole data

structure is considered to be affected by the error flow.

A data item is said to employ explicit addressing if it is a simple data item; otherwise, it is said to employ implicit addressing. An example of implicit addressing of data items is the array data structure. A control item is treated as employing explicit addressing although there is no memory address corresponding to it.

The characterization of the potential error behavior of a block can be formally defined as follows:

Definition 5.1. The block error characteristics of a block b consists of two sets $C[b]$ and $P[b]$, and a mapping $FM[b]$. The source capable set $C[b]$ of b is the set of items which can become error sources due to an execution of b . A subset of $C[b]$ consisting of the elements in $C[b]$ which employ explicit addressing is called the explicit source capable subset of b and denoted by $EC[b]$. The potential propagator set $P[b]$ of b is the set of items which can implicate some secondary error sources due to an execution of b . The flow mapping $FM[b]$ of b is a function from the set $P[b]$ to the power set of $C[b]$. For each element p of the set $P[b]$, the subset of $C[b]$ which is the image of p under the flow mapping $FM[b]$ is defined as

$$FM[b](p) = \{ c \in C[b] \mid p \text{ can implicate } c \text{ as a secondary error source due to an execution of } b \}.$$

The error characteristics of a local block characterize the potential error behavior of the statement or expression contained in the block. On the other hand, the error characteristics of a sequence of three external blocks for a module invocation characterize the potential error behavior of the module invocation.

It is clear that the potential propagator set and the source capable set are needed for modelling the potential error behavior of a block. The flow mapping, which provides the relationships between the two sets of items, is also needed because the sets of secondary error sources implicated by the elements of the potential propagator set can be different when the block is an external block used to model the potential error behavior of a module invocation, or when multiple assignment in a simple statement is possible for the source language of the program to be analyzed. Furthermore, the block error characteristics defined above are sufficient to characterize the potential error behavior of a block because the source capable set provides the set of items which can become an error source, and the potential propagator set and the flowing mapping together provide the set of items which can implicate some secondary error sources as well as their implicated secondary error sources due to an execution of the block.

5.1.1.2 Construction Of Intramodule Error Flow Model

The construction of the intramodule error flow model for a program is similar to the identification of the local data flow information in data flow analysis techniques [ALLE74], [LOME77], [BART78], [ROSE79], [ARTH81]. For a local block b , the source capable set $C[b]$ basically corresponds to the MODIFY set which is commonly used in data flow analysis techniques. This is because each definition x in b can become an error source either due to an initial program modification to the definition of x in b , or x is defined in b with some usages which are error sources. The potential propagator set $P[b]$ basically corresponds to the USE set in data flow analysis techniques because each usage y in b is used to define some definition x in b . Hence, y can implicate x as a secondary error source if y is an error source flowing into b . Note that the control definitions and usages are included in the block error characterization. Furthermore, the block error characteristics sets of the entry and exit blocks are specified as empty sets because they do not correspond to any executable statements.

In the intramodule error flow model construction process, only control usages are identified in the block error characteristics of the external blocks in the program. These block error characteristics will be updated when the

intermodule error flow model is constructed.

The intramodule error flow model can be constructed by an extended parser [AH072] of the source language of the program to be analyzed. The intramodule error flow model construction process can be described quite formally using an attributed grammar [AH072] of the source language.

Example 5.1. Consider the PASCAL program given in Figure 5.1. This program computes the roots of a quadratic equation $a*x*x + b*x + c = 0$. The block error characteristics of the blocks 1 to 6 which are constructed in the procedure roots are shown in Figure 5.2. The control flow in the procedure roots is sequential.

5.1.1.3 Intramodule Error Flow Tracing

Now, let us consider the tracing of the error flow within a module. The error flow can be described in terms of the propagation error source sets of the blocks as defined below.

Definition 5.2. The propagation error source set $ES[b]$ of a program block b consists of the set of error sources which reach the exit point of the block b .

Block Source code

```

program example(input, output);
var a, b, c, xr1, xr2, xi: real;

    procedure roots(aa, bb, cc: real);
    var x1, xr, xs, disc: real;

        procedure rroots(rrootsdisc, rrootsx1);
        var rrootsx2: real;
1       begin
2           rrootsx2 := sqrt(rrootsdisc);
3           xr1 := rrootsx1 + rrootsx2;
4           xr2 := rrootsx1 - rrootsx2;
5           xi := 0
6       end;

        procedure iroots(irootsdisc, irootsx1);
        var irootsx2: real;
7       begin
8           irootsx2 := sqrt(-irootsdisc);
9           xr1 := irootsx1;
10          xr2 := irootsx1;
11          xi := irootsx2
12      end;

13      begin
14          x1 := - bb / (2.0 * aa);
15          xr := x1 * x1;
16          xs := cc / aa;
17          disc := xr - xs;
18          if disc >= 0
19,20,21      then rroots(disc, x1)
22,23,24      else iroots(disc, x1)
25      end;

26      begin
27          read(a, b, c);
28          if a <> 0 then
                begin
29,30,31          roots(a, b, c);
32          writeln(xr1, xr2, xi)
                end
33          else writeln('Not a quadratic equation')
34      end.

```

Figure 5.1. An example program.

```

C[1] = 0;
P[1] = 0;

C[2] = { rootsx2 };
P[2] = { rootsdisc };
FM[2](rootsdisc) = { rootsx2 };

C[3] = { xr1 };
P[3] = { rootsx1, rootsx2 };
FM[3](rootsx1) = FM[3](rootsx2) = { xr1 };

C[4] = { xr2 };
P[4] = { rootsx1, rootsx2 };
FM[4](rootsx1) = FM[3](rootsx2) = { xr2 };

C[5] = { xi };
P[5] = 0;

C[6] = 0;
P[6] = 0;

```

Figure 5.2. The error characteristics of the blocks in roots in the program shown in Figure 5.1.

Given the error source set $ES[a]$ of a block a , the sets of error sources which reach the exit points of the immediate successor blocks of a can be determined based on the set $ES[a]$ and the block error characteristics of these immediate successor blocks. A tracing function $f(a, b)$ is defined below to derive the set of error sources which reach the exit point of a block b , given the propagation error source set $ES[a]$ of

an immediate predecessor block a of the block b . An error source x will flow out of block b as a result of the incoming error source set $ES[a]$ if one of the following two conditions holds:

- (1) x is implicated in b as a secondary error source by an element of $ES[a]$, or
- (2) x is an incoming error source which passes through b .

Thus, the tracing function $f(a, b)$ is defined as the union of the two sets of error sources, each of which contains all of the error sources satisfying one of the above two conditions. Under Condition (1), each element x in the intersection of $P[b]$ and $ES[a]$ is capable of implicating a set of secondary error sources in b because x is an incoming error source and it can propagate potential errors to some items in b . The set of secondary error sources implicated by x in b can be obtained by the flow mapping on x , i.e. $FM[b](x)$. Hence, $FM[b](P[b] \cap ES[a])$ is the set of all secondary error sources implicated in b by the incoming error source set $ES[a]$. Under Condition (2), an incoming error source x cannot pass through b if it employs explicit addressing and it is redefined in b . In other words, x cannot pass through b if it is an element of the explicit source capable subset $EC[b]$ of the block b . Hence, $(ES[a] - EC[b])$ is the set of incoming error sources which passes through b . Therefore, we have

$$f(a, b) = (ES[a] - EC[b]) \cup FM[b](P[b] \cap ES[a]).$$

The intramodule error flow from the points of initial program modification to other areas in the module can then be traced utilizing this tracing function along with the error characteristics of the module's blocks and the control flow graph of this module. The tracing function can be applied on a block-immediate successor block basis to form an algorithmic technique to trace the intramodule error flow. Applying the tracing function on a block-immediate successor block basis means that errors are propagated from an initial error source block s to all immediate successor blocks t of s , and then from t to all immediate successor blocks of t , etc. Application of the tracing function repeatedly in this manner identifies the propagation error source set $ES[i]$ of a block i in a stepwise manner with all the error sources flowing from an immediate predecessor block of i to i contributing to the final $ES[i]$. The tracing function is applied in this manner while new secondary error sources are identified.

This intramodule error flow tracing scheme can be formalized as an algorithm. It is assumed in this algorithm that the propagation error source sets in the module m are initialized according to some initial error flow condition. Also assumed is an initial error source block set $IB[m]$ of m consisting of the blocks in m which have non-empty initial

propagation error source sets. This algorithm is given below.

Algorithm 5.1. Intramodule Error Flow Tracing

Step 1. If $IB[m]$ is empty, then terminate. Otherwise, select an element from $IB[m]$ and then delete it from $IB[m]$. Let b denote the selected block.

Step 2. For each immediate successor block b' of b , first, check if $f(b, b')$ is a subset of $ES[b']$. If it is not, then let $ES[b'] = ES[b'] \cup f(b, b')$, and insert b' into $IB[m]$. After all the immediate successor blocks of b have been examined, go to Step 1.

The proof that algorithm 5.1 correctly identifies the intramodule error flow in m implicated by the initial propagation error source sets of the blocks in m can be found in [HSIE82]. Now, we would like to give an example to illustrate this algorithm.

Example 5.2. Consider the procedure roots in the program given in Figure 5.1. Assume the initial error flow in the procedure roots is given as $IB[rroots] = \{ 1 \}$, where $ES[1] = \{ rootsdisc \}$ and $ES[b] = \emptyset$ for the remaining blocks b in roots, i.e. the input parameter rootsdisc is the only error source in the procedure roots flowing out of the entry block 1 of roots. The intramodule error flow tracing for the procedure roots is then illustrated in Figure 5.3.

Input. ES[1] = { rootsdisc }.
ES[i] = \emptyset , for i = 2 to 6.

Step 1. Since IB[roots] = { 1 }, select block 1 from IB[roots], and let IB[roots] = \emptyset .

Step 2. Block 2 is the only immediate successor of block 1:

Since $f(1, 2) = \{ \text{rootsdisc}, \text{rootsx2} \}$ is not a subset of ES[2] = \emptyset , let ES[2] = { rootsdisc, rootsx2 }, and IB[roots] = { 2 }.

Step 1. Select block 2 from IB[roots], and let IB[roots] = \emptyset .

Step 2. Block 3 is the only immediate successor of block 2:

Since $f(2, 3) = \{ \text{rootsdisc}, \text{rootsx2}, \text{xr1} \}$ is not a subset of ES[3] = \emptyset , let ES[3] = { rootsdisc, rootsx2, xr1 }, and IB[roots] = { 3 }.

.
. .
. . .
. . .

Step 1. Select block 5 from IB[roots], and let IB[roots] = \emptyset .

Step 2. Block 6 is the only immediate successor of block 5:

Since $f(5, 6) = \{ \text{rootsdisc}, \text{rootsx2}, \text{xr1}, \text{xr2} \}$ is not a subset of ES[6] = \emptyset , let ES[6] = { rootsdisc, rootsx2, xr1, xr2 }, and IB[roots] = \emptyset .

Step 1. Select block 6 from IB[roots], and let IB[roots] = \emptyset .

Step 2. Since block 6 does not have any immediate successors, go to Step 1.

Step 1. Since IB[roots] = \emptyset , terminate.

The final propagation error source sets of the blocks in the procedure roots are given as follows:

ES[1] = { rootsdisc }.
ES[2] = { rootsdisc, rootsx2 }.
ES[3] = { rootsdisc, rootsx2, xr1 }.
ES[4] = { rootsdisc, rootsx2, xr1, xr2 }.
ES[5] = { rootsdisc, rootsx2, xr1, xr2 }.
ES[6] = { rootsdisc, rootsx2, xr1, xr2 }.

Figure 5.3. Intramodule error flow tracing in roots in the program shown in Figure 5.1.

The intramodule error flow model and the intramodule error flow tracing scheme together model and trace the potential error flow within a module.

5.1.2 Intermodule Error Flow Model

In this section, the intermodule error flow model is presented. The intermodule error flow model models how potential error sources can propagate between the modules in the program. Error sources can propagate between the invoking and invoked modules through parameter passing or data sharing via module invocation.

A program can be considered as a collection of program modules. There exists one and only one module in the program which starts program execution upon invocation by the operating system. This module is called the main module. Upon invocation, a module is executed and then the module returns control to the invoking module at the invocation site upon exit from the module. The invocation relationships among the modules in the program can be represented by a call graph of the program [ALLE74].

Recently, much effort has been devoted to the development

of intermodule data flow analysis techniques with applications primarily to compiler optimization and static program analysis [ALLE74], [LOME77], [BART78], [ROSE79], [ARTH81]. Intermodule data flow information that is used at the point of module invocation has been called summary data flow information [ALLE74]. With each module invocation, a summary of the variables which may be modified, used, or preserved due to this module invocation will be generated for data flow analysis.

In our logical ripple effect analysis, the intermodule error flow is modelled utilizing an approach similar to usual intermodule data flow analysis techniques. The summary error flow information of a module is called the module error characteristics of the module, and will be generated to represent the potential error flow properties of the module.

In order to model the intermodule error flow which occurs at a module invocation, a sequence of three blocks is constructed in the invoking module for this invocation. The first block in the sequence, called an input parameter mapping block, is used to establish the error flow from the actual input parameters to their corresponding formal input parameters of the invoked module. The second block in the sequence, called an invocation block, is used to reflect the potential error flow properties of the invoked module represented by the

module error characteristics of the invoked module. The third block in the sequence, called an output parameter mapping block, is used to establish the error flow from the formal output parameters of the invoked module to their corresponding actual output parameters. The error characteristics of the three blocks can be updated, after the error characteristics of the invoked module have been generated, based on the invoked module's error characteristics and the parameter passing information associated with this invocation.

5.1.2.1 Module Error Characteristics

To define the module error characteristics of a module m , it is first necessary to identify the data interface of m consisting of the items which can interact with the global environment of m . The data interface of m is represented by the parameter set of m which is formally described by the following definition:

Definition 5.3. The parameter set $PS[m]$ of a module m consists of the formal parameters of m , the item representing the return value of the module if m is a function, and the data items which are global to m and are referenced in m or any of m 's invoked modules.

The module error characteristics of a module m can be formally defined as follows:

Definition 5.4. The module error characteristics of a module m consists of two sets $MC[m]$ and $MP[m]$, and a mapping $MFM[m]$. The module source capable set $MC[m]$ of m consists of the items in the parameter set $PS[m]$ of m each of which can become an error source due to an invocation of m . The module potential propagator set $MP[m]$ of m consists of the elements in $PS[m]$ each of which can implicate some elements in $PS[m]$ as secondary error sources capable of affecting the global environment of m due to an invocation of m . The module flow mapping $MFM[m]$ of a module m is a function from the set $MP[m]$ to the power set of $MC[m]$. For each element p of the set $MP[m]$, the subset of $MC[m]$ which is the image of p under $MFM[m]$ is defined as

$$MFM[m](p) = \{ c \in MC[m] \mid p \text{ can implicate } c \text{ as a secondary error source due to an invocation of } m \}.$$

The module error characteristics of a module m provide the set of items which can become error sources capable of affecting the global environment of m , and the set of items which can propagate potential error sources from the global environment of m to implicate some secondary error sources capable of affecting the global environment of m as well as their implicated secondary error sources due to an invocation of m . It is clear that the module error characteristics of a

module are necessary for modelling the potential error flow behavior of the module. To show that they are sufficient, it is not necessary to include the items which are not elements of the parameter set $PS(m)$ of a module m in the module error characteristics of m because they cannot interact with the global environment of m . Furthermore, it is not necessary to include an item x , capable of implicating some elements in $PS(m)$ as secondary error sources none of which can affect the global environment of m , in the module propagator set of m because the error sources implicated by x cannot affect the global environment of m . Therefore, the module error characteristics are sufficient to model the potential error flow properties of a module.

The order in which the error characteristics of the modules in the program are generated is very important because the error characteristics of an invoked module can affect those of its invoking modules. In nonrecursive programs, there is some ordering, called the reverse invocation order which has the property that when modules are examined in this order, invoked modules are always analyzed in advance of the modules which invoke them [ALLE74]. Therefore, the error characteristics of the modules in a nonrecursive program can be generated following the reverse invocation order. In the case of recursive programs, there is no ordering with such a

property. Furthermore, the local variables of a recursive module may exhibit different error flow properties in different activations of the module because recursive activations of the module will create separate copies of the module's local variables, called incarnations of the variables.

Intermodule error flow is also complicated by dynamic aliasing, which is a problem that occurs when syntactically distinct names are used to represent the same or overlapping storage areas at run time. In the presence of dynamic aliasing, the error characteristics of the modules must be generated with the consideration of dynamic aliasing conditions. Dynamic aliasing can be caused by reference parameters.

Our approach to model the intermodule error flow for nonrecursive programs which do not have any dynamic aliasing anomalies will be described here. By a dynamic aliasing anomaly we refer to the problem where either a variable is passed by reference to more than one formal parameter of a module, or a global variable which is referenced in a module is also passed by reference to a formal parameter of that module. In the presence of a dynamic aliasing anomaly, the formal reference parameters of the module cannot be treated as independent entities. Dynamic aliasing anomalies tend to complicate testing of programs, and hence modern programming

practices advocate the elimination of dynamic aliasing anomalies [WASS80], [ICHB79]. An approach to handle programs which have recursion or dynamic aliasing anomalies can be found in [HSIE82].

For a nonrecursive program without any dynamic aliasing anomaly, the error characteristics of the modules in the program can be generated following the reverse invocation order. Each module has to be analyzed only once. After the error characteristics of a module have been generated, the error characteristics of the external blocks for invocations of the module are then updated.

5.1.2.2 Generation Of Module Error Characteristics

The error characteristics of a module m can be generated by the following algorithm based on the parameter set $PS[m]$ of m and the error characteristics of all the blocks in m .

Algorithm 5.2. Identification of Module Error Characteristics

Step 1. Initialize the set $MP[m]$ to be empty.

Step 2. Calculate the set $MC[m]$ by computing $(PS[m] \cap (\cup C[b] \mid b \text{ is a block in } m))$.

Step 3. Obtain a set T by computing $(PS[m] \cap (\cup P[b] \mid b \text{ is a block in } m))$.

Step 4. If T is empty, then terminate. Otherwise, select an

element from T , and then delete it from T . Let x denote the selected element.

Step 5. Let $IB[m] = \{u\}$, where u is the entry block of m . Let $ES[u] = \{x\}$, and $ES[b] = \emptyset$ for the rest of blocks in m .

Step 6. Apply Algorithm 5.1 to trace the intramodule error flow in m .

Step 7. Check if $(ES[v] \cap MC[m])$ is empty, where $ES[v]$ is the propagation error source set of the exit block v of m . If it is not, then insert x into the set $MP[m]$, and let $MFM[m](x) = (ES[v] \cap MC[m])$. Go to Step 4.

The proof that Algorithm 5.2 correctly identifies the error characteristics of a module is given in [HSIE82]. Now, we would like to give an example to illustrate this algorithm.

Example 5.3. Consider the procedure roots given in Figure 5.1. The parameter set $PS[\text{roots}]$ of the procedure roots can be easily identified as $PS[\text{roots}] = \{\text{rootdisc}, \text{rootx1}, \text{xr1}, \text{xr2}, \text{xi}\}$. The identification of the module error characteristics of the procedure roots is illustrated in Figure 5.4.

- Step 1. $MP[rroots] = \emptyset$.
- Step 2. $MC[rroots] = \{ rootsdisc, rootsx1, xr1, xr2, xi \}$
 $\{ rootsx2, xr1, xr2, xi \}$
 $= \{ xr1, xr2, xi \}$.
- Step 3. $T = \{ rootsdisc, rootsx1, xr1, xr2, xi \}$
 $\{ rootsdisc, rootsx1, rootsx2 \}$
 $= \{ rootsdisc, rootsx1 \}$.
- Step 4. Select $rootsdisc$ from T , and then let $T = \{ rootsx1 \}$.
- Step 5. Let $IB[rroots] = \{ 1 \}$, $ES[1] = \{ rootsdisc \}$, and
 $ES[i] = \emptyset$, for i from 2 to 6.
- Step 6. The set $ES[6]$ obtained by Algorithm 1 is
 $ES[6] = \{ rootsdisc, rootsx2, xr1, xr2 \}$.
- Step 7. $ES[6] \quad MC[rroots] = \{ xr1, xr2 \}$.
Therefore, let $MP[rroots] = \{ rootsdisc \}$, and
 $MFM[rroots](rootsdisc) = \{ xr1, xr2 \}$.
- Step 4. Select $rootsx1$ from T , and then let $T = \emptyset$.
- Step 5. Let $IB[rroots] = \{ 1 \}$, $ES[1] = \{ rootsx1 \}$, and
 $ES[i] = \emptyset$, for i from 2 to 6.
- Step 6. The set $ES[6]$ obtained by Algorithm 1 is
 $ES[6] = \{ rootsx1, xr1, xr2 \}$.
- Step 7. $ES[6] \quad MC[rroots] = \{ xr1, xr2 \}$.
Therefore, let $MP[rroots] = \{ rootsdisc, rootsx1 \}$,
and $MFM[rroots](rootsx1) = \{ xr1, xr2 \}$.
- Step 4. Since T is empty, terminate.

The error characteristics of the procedure $roots$ identified by Algorithm 2 are as follows:

$MC[rroots] = \{ xr1, xr2, xi \};$
 $MP[rroots] = \{ rootsdisc, rootsx1 \};$
 $MFM[rroots](rootsdisc) = MFM[rroots](rootsx1)$
 $= \{ xr1, xr2 \}.$

Figure 5.4. The module error characteristics of $roots$ in the program shown in Figure 5.1.

5.1 2.3 Update Block Error Characteristics

Let $i, j,$ and k be a sequence of three external blocks in a module n for an invocation of m . For the first block i in the sequence, i.e. the input parameter mapping block, each formal input parameter x of m should be inserted into the source capable set $CC[i]$ of block i . Each data item y which has positional correspondence to x in the actual parameter list of this invocation should be inserted into the potential propagator set $PI[i]$ of block i , while x is inserted into $FMI[i](y)$.

For the second block j in the sequence, i.e. the invocation block, each element of the module source capable set $MC[m]$ of m should be inserted into the source capable set $CC[j]$ of block j . Also, each element x of the module potential propagator set $MP[m]$ of m should be inserted into the potential propagator set $PI[j]$ of block j , while each element of $MFMI[m](x)$ is inserted into $FMI[j](x)$.

For the last block k in the sequence, i.e. the output parameter mapping block, each formal output parameter z of m should be inserted into the potential propagator set $PI[k]$ of block k . For each formal output parameter z of m , let w be the data definition and X be the set of usages in the actual

parameter list associated with this invocation which have positional correspondence to z . Then, w should be inserted into both the source capable set $C[k]$ of block k and $FM[k](z)$. Furthermore, each element x of the set X should be inserted into the potential propagator set $P[k]$ of block k , while w is inserted into $FM[k](x)$.

Furthermore, for each control usage in the potential propagator set of each block in the sequence identified by the intramodule error flow model construction process, the flow mapping on the control usage is updated to be the source capable set of the block.

For some programming languages, such as JOVIAL, a formal parameter of a module can be identified as an input or output formal parameter based on the syntax rules of the languages. For other programming languages which cannot distinguish syntactically between the formal input and output parameters, a formal parameter x of a module m is an input formal parameter if x is an element of the module potential propagator set $MP[m]$ of m . A formal parameter y of m is an output formal parameter if y is an element of the module source capable set $MC[m]$ of m .

Example 5.4. Consider the invocation of the procedure `roots` in the program shown in Figure 5.1. Blocks 19 to 21 are the external blocks constructed for this invocation. Let $c.1$

denote the control item representing the predicate ($\text{disc} \geq 0$). The block error characteristics of the three blocks identified by the intramodule error flow construction process are given as follows:

```

C[19] = 0; P[19] = { c.1 }; FM[19](c.1) = 0;
C[20] = 0; P[20] = { c.1 }; FM[20](c.1) = 0;
C[21] = 0; P[21] = { c.1 }; FM[21](c.1) = 0;

```

Based on the module error characteristics of the procedure roots described in Figure 5.4, the error characteristics of the three blocks can be updated as follows:

```

C[19] = { rootsdisc, rootsx1 };
P[19] = { disc, x1, c.1 };
FM[19](disc) = { rootsdisc }; FM[19](x1) = { rootsx1 };
FM[19](c.1) = C[19].

```

```

C[20] = { xr1, xr2, xi };
P[20] = { rootsdisc, rootsx1, c.1 };
FM[20](rootsdisc) = FM[20](rootsx1) = { xr1, xr2 };
FM[20](c.1) = C[20].

```

```

C[21] = 0; P[21] = { c.1 };
FM[21](c.1) = C[21].

```

5.1.3 Logical Ripple Effect Identification

In this section, the identification of the logical ripple effect of an initial program modification is described. The logical ripple effect can be identified in two steps. The first step is the error flow tracing step which traces the error flow in the modified program implicated by the primary error sources. The second step is the logical ripple effect derivation step which derives the logical ripple effect of the

initial program modification based on the error flow in the program.

5.1.3.1 Error Flow Tracing

The error flow tracing requires the tracing of error flow both within modules and between modules. Potential error sources can propagate from a module m to the modules which invoke m , and to the modules which are invoked by m . When there exists error flow from module m to the modules invoked by m , error sources are said to propagate in a downward direction with respect to module m . Similarly, when there exists error flow from module m to the modules which invoke m , error sources are said to propagate in an upward direction with respect to m . It is apparent that the downward intermodule error flow with respect to m must be identified before the upward intermodule error flow with respect to m is identified; otherwise, the latter cannot be completely characterized.

Let PRIMESET be the primary error source set of a program, in which each element (m, b, x) denotes that x is a primary error source at block b in module m . The error flow tracing identifies the modules, blocks, and items which are implicated by the error flow caused by the primary error source set PRIMESET.

5.1.3.2 Intermodule Error Flow Tracing

The existence of the upward intermodule error flow from a module n can be identified as follows: A module n can propagate error sources upward to each module which invokes n via each invocation of n if and only if $(MC[n] \cap ES[v]) \neq \emptyset$, where $ES[v]$ is the propagation error source set of the exit block v of n and $MC[n]$ is the module source capable set of n . The elements of $(MC[n] \cap ES[v])$ are used to update the propagation error source set of each invocation block constructed for an invocation of n such that the error flow implicated by these upward intermodule error sources can be traced.

The presence of the downward intermodule error flow from a module m to an invoked module n via an invocation of n can be identified as follows: Suppose that a module m is invoked in a module n and b is the input parameter block constructed in n for this invocation. Given the propagation error source set $ES[b]$ of b , n can propagate potential errors to m via this invocation if $(ES[b] \cap MP[m]) \neq \emptyset$, where $MP[m]$ is the module potential propagator set of m . The elements of $(ES[b] \cap MP[m])$ are used to update the propagation error source set of the entry block of module m such that the error flow implicated by these downward intermodule error sources can be traced.

5.1.3.3 Error Flow Tracing Algorithm

The areas in a program which are implicated by the error flow in the program is identified in a stepwise manner. The primary error source set PRIMESET is used to initialize the propagation error source sets and the initial error source block sets. The intramodule error flow in the modules involved in initial modification is then traced based on the initial propagation error source sets of the blocks in the modules. After the intramodule error flow in a module m stabilizes, the intermodule error flow originating at m implicated by the error flow is then identified based on these propagation error source sets, and used to update the propagation error source sets of the blocks in the modules to which the intermodule error flow is propagated. The modules which are implicated by the intermodule error flow are then analyzed. This process continues until the error flow stabilizes, i.e. no new error sources are identified.

An algorithm has been developed for identifying the program areas which are implicated by the error flow caused by the primary error source set PRIMESET. Let AFFECTM be the set of modules which are implicated by the error flow. In this algorithm, a set UPM is used to contain the modules potentially affected by the upward intermodule error flow, and a set DOWNM is used to contain the modules potentially affected by the

downward intermodule error flow. This algorithm is given below.

Algorithm 5.3. Error Flow Tracing

Step 1. Initialize the sets AFFECTM and UPM all to be empty. For each module m in the program, initialize the set $IB[m]$ to be empty. For each block b in the program, initialize the set $ES[b]$ to be empty.

Step 2. For each element (m, b, x) of the set PRIMESET, insert x into the set $ES[b]$. Furthermore, insert b into $IB[m]$, and m into the sets AFFECTM and UPM.

Step 3. If UPM is empty, then terminate. Otherwise, select a module from UPM and delete it from UPM. Let n denote the selected module.

Step 4. Identify the intramodule error flow in n utilizing Algorithm 5.1.

Step 5. Calculate $T = (ES[v] \cap MC[n])$, where v is the exit block of n . If T is not empty, then for each invocation block b in a module k constructed for an invocation of n , check if T is a subset of the propagation error source set $ES[b]$ of b . If it is not, i.e. new error sources flow out of n upward to k via this invocation, then insert k into AFFECTM and UPM, and b into $IB[k]$. Furthermore, let $ES[b] = (ES[b] \cap T)$.

Step 6. Let $DOWNM = \emptyset$. Then, for each input parameter mapping block b in n for an invocation of some module m , calculate a

set $T = (ES[b] \cap MP[m])$. Check if T is a subset of the propagation error source set $ES[u]$ of the entry block u of module m . If it is not, i.e. new error sources flow into m , then insert m into $AFFECTM$ and $DOWNM$, and u into $IB[m]$. Furthermore, let $ES[u] = (ES[u] \cup T)$.

Step 7. If $DOWNM$ is not empty, then select a module from $DOWNM$ and delete it from $DOWNM$. Let j denote the selected module. Repeat Steps 4 and 6 with j substituting n to trace the intramodule error flow in j and the downward intermodule error flow propagated from j . This process continues until the set $DOWNM$ becomes empty. i.e. the error flow implicated by the downward intermodule error flow originating at n stabilizes. Then go to Step 3 to trace the error flow implicated by the upward intermodule error flow from the modules in the set UPM .

The proof that Algorithm 5.3 correctly identifies the areas in a program which are implicated by the error flow in the program caused by the primary error source set $PRIMESET$ is given in [HSIE82]. Now, let us give the following example to illustrate this algorithm.

Example 5.5. Consider the program shown in Figure 5.1. Assume that the initial modification corrected the definition of xi in the procedure $roots$, i.e. the primary error source set of the program is given by $PRIMESET = \{ (roots, 5, xi) \}$. The error flow tracing by Algorithm 5.3 is illustrated in Figure 5.5.

- Step 1. Let AFFECTM = \emptyset , and UPM = \emptyset .
Let ES[i] be empty, for each block i.
- Step 2. Let ES[5] = { xi }, IB[roots] = { 5 }, and
UPM = { roots }, AFFECTM = { roots }.
- Step 3. Select roots from UPM, and then let UPM = \emptyset .
:
.
- Step 5. Since (ES[6] MC[roots]) = { xi }, let
UPM = { roots }, AFFECTM = { roots, roots },
ES[20] = { xi }, and IB[roots] = { 20 }.
- Step 6. Since roots has no immediate successors,
go to Step 3.
:
.
- Step 5. Since (ES[25] MC[roots]) = { xi }, let
UPM = { example }, IB[example] = { 30 },
AFFECTM = { roots, roots, example }, and
ES[30] = { xi }.
- Step 6. Since no downward intermodule error flow from roots,
go to Step 3.
- Step 3. Select example from UPM, and then let UPM = \emptyset .
:
.
- Step 5. Since no upward intermodule error flow from example,
UPM = \emptyset .
- Step 6. Since no downward intermodule error flow from example,
go to Step 3.
- Step 3. Since UPM is empty, terminate.

The result of error flow tracing is as follows:

AFFECTM = { example, roots, roots };

Figure 5.5. Error flow tracing in the program shown in
Figure 5.1.

5.1.3.4 Logical Ripple Effect Derivation

We will use $RIPPLE[b]$ to denote the set of items in a block b which are affected by the logical ripple effect, $RIPPLEB[m]$ the set of blocks in a module m which are affected by the logical ripple effect, and $RIPPLEM$ the set of modules in a program which are affected by the logical ripple effect.

To derive the logical ripple effect from the error flow, it is first observed that a block may not be affected by the logical ripple effect even though the propagation error source set resulted from the error flow is not empty. This can be true if the elements in the propagation error source set are error sources which just pass through this block. Furthermore, it can easily be shown that an item x is affected by the logical ripple effect in block b only if $x \in (ES[b] \cap C[b])$.

Given the set $AFFECTM$ and the propagation error source set, which are derived in the error flow tracing step, the first step in logical ripple effect derivation is as follows: For each module m in the set $AFFECTM$, first initialize $RIPPLEB[m]$ to be an empty set. Then, for each block b in m , check if $(ES[b] \cap C[b])$ is empty. If it is not, then let $RIPPLE[b] = (ES[b] \cap C[b])$, and insert b into $RIPPLEB[m]$.

Next, it is observed that a module m may not be affected by the logical ripple effect despite the fact that m is implicated by the error flow. This can happen when all the error sources in m are just passing through m to the modules invoked by m without internally generating error sources in m . It is obvious that a module m is not affected by the logical ripple effect, if all the blocks in the set $RIPPLEB[m]$ are external blocks. Therefore, the next step in the logical ripple effect derivation is to identify the subset $RIPPLEM$ of the modules in the set $AFFECTM$ which have at least one local block in their $RIPPLEB[m]$ sets.

In an analogous manner, a block b with a nonempty set $RIPPLE[b]$ may not be affected by the logical ripple effect, if the block is an external block constructed for an invocation of a module which is not an element of the set $RIPPLEM$. Therefore, the final step in the logical ripple effect derivation is, for each module m in the set $RIPPLEM$, to remove the blocks b in the set $RIPPLEB[m]$ which are constructed for invocations of modules which are elements of the set $(AFFECTM - RIPPLEM)$.

Now, the set RIPPLEM gives the set of modules in a program which are affected by the logical ripple effect. For each module m in the set RIPPLEM, the set RIPPLEB[b] gives the set of blocks in m which are affected by the logical ripple effect. For each block b in the set RIPPLEB[m], the set RIPPLE[b] gives the set of error sources in b which may cause logical inconsistencies with the initial modification.

Example 5.6. Consider the program shown in Figure 5.1. The result of error flow tracing in the program has been shown in Figure 5.5. Since the only block in the procedure roots is an external block, the procedure roots is not affected by the logical ripple effect. Hence, the procedure roots is not included in the set RIPPLEM. Furthermore, block 30 in the main module example is constructed for an invocation of the procedure roots which is not affected by the logical ripple effect. Hence, block 30 is eliminated from the set RIPPLEB[example]. The logical ripple effect is thus given as follows:

RIPPLEM = { example, roots }.

RIPPLEB[example] = { 32 }.

RIPPLEB[roots] = { 5 }.

RIPPLE[32] = { output }.

RIPPLE[5] = { xi }.

In this section, a scheme to identify the logical ripple effect based on the set of primary error sources has been presented. Note that this scheme illustrates the concept of logical ripple effect identification. A more efficient algorithm can be found in [HSIE82].

The intramodule error flow model, the intermodule error flow model, and the logical ripple effect identification scheme together provide a model based on which the logical ripple effect can be identified. In the next section, the overall logical ripple effect analysis technique will be presented.

5.1.4 Logical Ripple Effect Analysis Technique

The logical ripple effect analysis technique can now be summarized as follows:

Step 1. Construct the intramodule error flow model as described in Section 5.1.1.

Step 2. Construct the intermodule error flow model as described in Section 5.1.2.

Step 3. Identify the primary error source set PRIMESET based on the initial program modification.

Step 4. Identify the logical ripple effect of the initial program modification as described in Section 5.1.3.

Steps 1, 2, and 4 of the logical ripple effect analysis technique can be automated without difficulty. However, the identification of primary error sources is more complicated, and the automation of this process is not simple. We will now discuss this step in more detail.

The primary error sources are identified to transform the initial program modification into the changes to the error flow of a program. To illustrate the identification of the primary error sources, let us consider the following types of initial program modifications:

(1) Suppose that a control condition was modified by changes to the data usages, relational operators, or constants in this control condition. The control definition associated with this control condition is then specified as a primary error source at the block which contains the control condition.

(2) Suppose that a data definition was changed or added in a block. The definition is then specified as a primary error source at the block.

(3) Suppose that a data definition was deleted. The definition is then specified as a primary error source at the block to which the original definition transferred control. Furthermore, if any definition in the block is defined with a usage of the deleted data definition, then the definition is also specified as a primary error source at the block.

(4) Suppose that an actual parameter x was replaced by y in a module invocation. If the corresponding formal parameter f is an input parameter, then f is specified as a primary error source at the input parameter mapping block for this invocation. If f is an output parameter, then x and y are both specified as primary error sources at the output parameter mapping block for this invocation.

(5) Suppose that a module invocation which invokes a newly added or an existing module was inserted into the program. The elements of the module source capable set of the invoked module are then specified as primary error sources at the invocation block for this newly added module invocation.

(6) Suppose that a module invocation n was deleted from a module m . The elements of the module source capable set of the invoked module with the formal output parameters substituted by their corresponding actual parameters in the deleted module invocation are then specified as primary error sources at the block to which the deleted module invocation transferred control.

(7) Suppose that an unconditional goto statement $s1$ which branches to a statement $s2$ was deleted from the program. Let $s3$ be the statement which followed the statement $s1$ in the original program. The data definitions which could reach $s2$ before the deletion of $s1$ but cannot reach $s2$ after the deletion of $s1$ are identified as primary error sources flowing

into the statement s2. Furthermore, the data definitions which could not reach s3 before the deletion of s1 but can after the deletion of s1 are identified as primary error sources flowing into the statement s3.

Our current logical ripple effect analysis technique requires the maintenance programmers manually identify the primary error sources. Further work is needed to automate this process.

5.1.5 Experiments

A prototype system to perform logical ripple effect analysis on PASCAL programs has been developed. This system consists of three subsystems: an intramodule error flow analyzer, an intermodule error flow analyzer, and a logical ripple effect identification subsystem. The identification of primary error sources should be performed manually by the maintenance programmers.

The intramodule error flow analyzer is developed by modifying an existing standard PASCAL compiler, while the other two subsystems are newly developed. The prototype system is currently running on a DEC VAX-11/780 computer under the VMS operating system. The system is primarily written in VAX-11 PASCAL, while some file handling routines are written in VAX-11

FORTRAN. The intramodule error flow analyzer and the intermodule error flow analyzer are run in batch mode, while the logical ripple effect identification subsystem can be run in either batch or interactive mode. The program sizes of the intramodule error flow analyzer, intermodule error flow analyzer, and logical ripple effect identification subsystem are 643, 190, and 230 disc blocks, respectively, where each disc block under the UMS operating system consists of 512 bytes.

During the logical ripple effect identification step, the user can specify the modules whose internal error flow will not be traced. For such a module, the upward error flow originating at this module will still be traced, but the downward error flow originating at this module will not be traced. Also, during interactive logical ripple effect identification, the user can remove an item from the error flow at a block such that further error flow implicated by this item would not be traced. This feature enables the user to control the scope of error flow tracing. For example, he can choose to trace only the intramodule error flow of a module which is involved in an initial program modification. Also, it can be used to reduce the scope of error flow tracing, and hence provide the user with more precise information about the potential logical inconsistencies. One example of a module

which is not traced can be an output routine which converts a data item from one format to another, while the routine itself is not modified. There are certain messages displayed on the terminal which can help the user better understand the error flow in the program implicated by the initial program modification.

We have applied our logical ripple effect analysis technique on PASCAL programs with sizes ranging from about 50 to 5000 lines of program statements and declarations. Based on our experiments, the execution time needed for the error flow analysis of a program depends on the program size. However, the response time for the interactive logical ripple effect identification is not significantly affected by the program size, but by the size and complexity of the modules in the program because the logical ripple effect identification is performed on a module-by-module basis.

Our experiment indicates that our logical ripple effect analysis technique can be very effective for scientific programs, which require extensive numerical computation. The logical ripple effect of an initial program modification follows very closely the data flow in this type of program. For other types of program the effectiveness of this technique is limited by the underlying data flow analysis technique. For example, since the data flow analysis cannot distinguish

distinct components of a complex data structure, the whole data structure is treated as modified if a particular component in the data structure is modified. This implies that all the program blocks which use different components of the data structure would be identified as affected by the ripple effect of the modification to a particular component in the data structure.

5.1.6 Discussion And Future Work

Our current logical ripple effect analysis technique requires the maintenance programmer manually identify the primary error sources and requires the program to be reanalyzed after each initial program modification to construct the error flow model of the modified program. The efficiency and ease of use of the logical ripple effect analysis technique can be improved by developing a scheme which can incrementally update the error flow model of the program and a scheme which can automatically identify the primary error sources.

The error flow model of a program can be incrementally updated in two steps: updating the intramodule error flow model and the intermodule error flow model. Since the intramodule error flow model can be constructed by an extended parser of the source language, the changes to the intramodule error flow model can be identified by an extended incremental

attributed grammar evaluator [DEME81] which performs incremental attribute reevaluation. An incremental attribute grammar evaluator can function together with a syntax-directed editor which can incrementally reevaluate the syntactic information of the program. The intermodule error flow model can then be incrementally updated by modifying the construction step of the intermodule error flow model to eliminate the analysis of the module error characteristics of a module if the module and each successor of the module is not involved in the initial program modification.

5.2 The Performance Ripple Effect Analysis Technique

Since a large-scale program usually possesses both functional and performance requirements, the ripple effect of program modifications must be analyzed from both a functional and a performance point of view. In many large-scale programs, the violation of a performance requirement is equivalent to a system error and thus requires further corrective action [BOYD78], [WEGN78], [SWAN76], [BELF77]. Consequently, in the maintenance process it is important 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 may be affected indirectly. The change in performance of these parts may then

have an impact on the performance of the other parts of the system.

In the previous contract, we developed a performance ripple effect analysis technique which was reported in detail in [YAUB80c, B0f]. This technique is based the identification of performance attributes, critical sections, performance propagation mechanisms, interdependency relationships among modules as well as the relations between user performance requirements and module performance attributes. Algorithms for identifying those items have been established and an algorithm for tracing the performance ripple effect has been established. During this project period, we have constructed a prototype system for the demonstration of our performance ripple effect analysis technique. In the following section, we will discuss our experimental results.

5.2.1 Experimentation

This prototype system, which has been developed to demonstrate our performance ripple effect analysis for PASCAL [JENS74] programs, is made up of two subsystems: a program text analyzer, which constructs a model of the program for tracing performance ripple effects, and a performance ripple effect tracing subsystem. Since PASCAL programs involve no concurrent operations, not all of the performance attributes,

critical sections, virtual performance attributes and the relationships among them could be shown. Therefore, the program text analyzer constructs only those portions of the model which are relevant to PASCAL programs, although the subsystem to trace the effects of program changes can also trace these effects on programs which include those portions of the model which are associated with concurrent operations.

The program text analyzer was developed by modifying an existing PASCAL compiler and consists of over 7000 lines of PASCAL code, while the tracing subsystem was newly developed and consists of about 2500 lines of PASCAL code. The prototype system is currently running under the VMS operating system on a DEC VAX-11/780 computer. This system is written entirely in VAX-11 PASCAL. Both subsystems run without user interaction, the first constructing the performance ripple effect model of the program, and the second tracing the effects of a modification through the entire program.

Since the logical correctness of a software system is at least as important as its ability to meet performance requirements, we will assume that an analysis of logical ripple effects precedes that of performance ripple effects. This allows us to take advantage of the data flow analysis performed by the logical ripple effect identification subsystem.

To assist us in validating the results of our performance ripple effect analysis we developed a technique to estimate the execution time of arbitrary paths in the programs being modified. This technique was described in [YAUB1b]. We compared the estimated execution times of all critical sections of the program, before and after the modification, and observed that all quantitative changes in estimated times appeared in critical sections that were implicated by our performance ripple effect analyzer.

5.2.2 Discussion

During the early stages of the maintenance process, the performance ripple effect analysis technique can be used as an aid in developing criteria for maintenance personnel to evaluate proposed program modifications from a performance perspective. Basically, this involves the worst-case identification of performance requirements which might be affected by the program modifications.

After a program modification has been selected and completely implemented, the performance ripple effect analysis technique can substantially refine its analysis and determine more accurately which performance requirements may have been affected by the program modifications. These performance requirements can then become the targets for retesting. 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 x and y , performance attribute y does not need to be examined for changes if it has been determined that performance attribute x is not affected by the maintenance activity. Thus, the preliminary results of some of the early retesting efforts may be decisive in determining the scale of retesting which remains to be done. The use of program assertions concerning the execution time of performance attributes would play an important role in determining if a performance attribute has been affected by a modification.

6.0 EFFECTIVE TESTING FOR SOFTWARE MAINTENANCE

Despite the use of automated tools to assist the maintenance programmer in making modifications correctly, the possibility of error remains, and so the modified program must still be retested. Nonetheless, we would like to avoid retesting the entire program if only a minor modification has been made.

We have developed a module testing approach which makes use of existing test cases whenever possible, and uses the input partition method [RICH81] for constructing new test cases when they are required. Actual testing is done by symbolic execution [KING76], but we make use of real test case data to select the control flow paths to be executed. We have demonstrated an implementation of our approach using ANSI FORTRAN by modifying the ATTEST system [CLAR76]. Our method is effective in testing programs with mathematical computations whose specifications can be given in the cause/effect manner [MYER76], [GOOD75], [HALL78], [HENI80], such as control programs for aircraft control systems and nuclear power plant control systems. Although our method has been demonstrated for programs written in FORTRAN, it can easily be modified for programs written in block-structured languages, such as PASCAL, PL/1 and ALGOL. The application of this method will also be discussed.

6.1 The Module Revalidation Technique

In this section, we will present our module revalidation technique during the maintenance phase. The smallest unit in the program we consider here as a modified section is a program section which is a maximal set of ordered statements of a program that can only be executed as follows: its execution starts from the first statement, terminates at the last statement, and all of its statements are executed in the given order. In addition, we assume that each module in the program has one entry and multiple exits. Our technique is applied only after all the necessary modifications of the module are completed. It is assumed that the module before the modification has been tested by the test set $T = \{t_1, t_2, \dots, t_n\}$, where T was generated by any test generation method, each test case $t_i = \{v_1, v_2, \dots, v_m\}$, $i=1, 2, \dots, n$, and v_j , $j=1, 2, \dots, m$, is an input value for the j th input variable of the module. Furthermore, we assume that the specification of each module is correct and given in the form of a cause/effect graph [MYER76].

The module revalidation technique can be summarized in the flow-chart shown in Figure 6.1. To start with, the derivation of the input partition for the modified module will be done to reflect the changes in the program code and/or specification. Then, the original test cases of T which are still correct

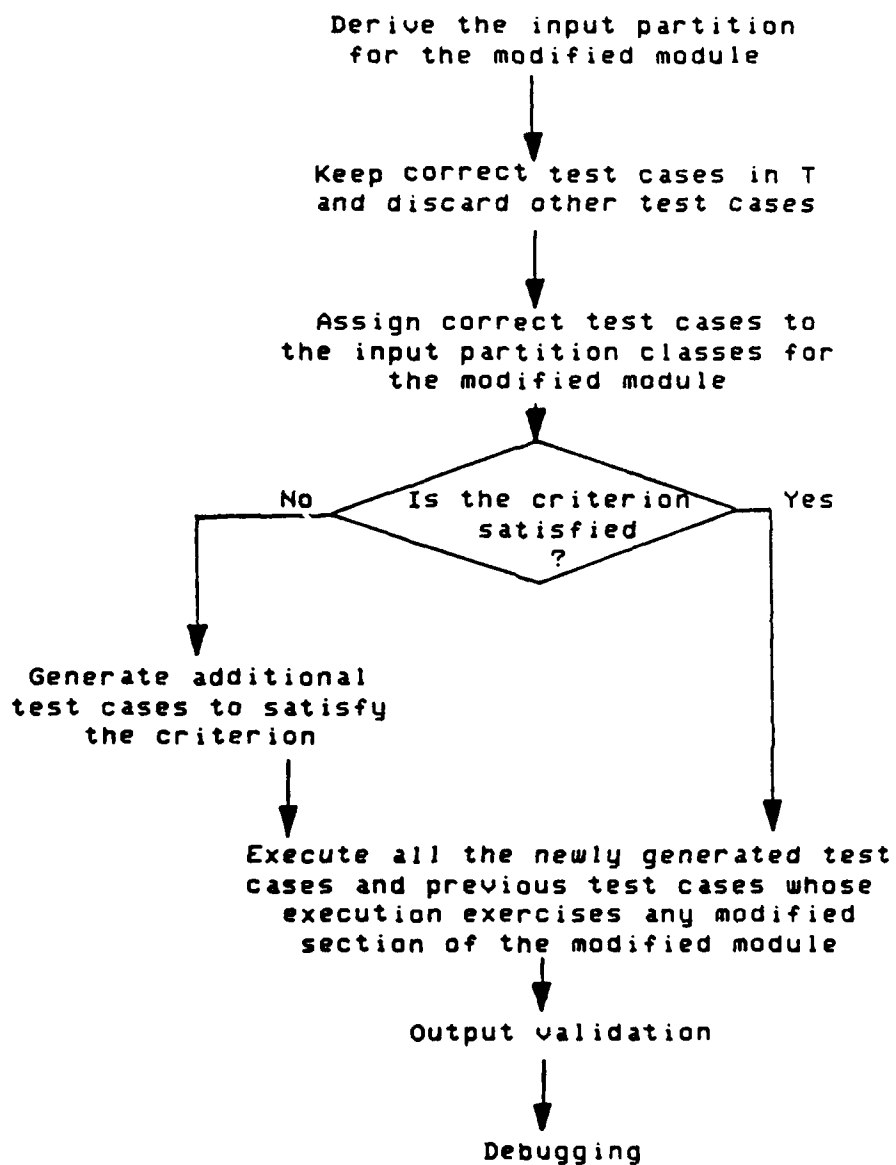


Figure 6.1 An overview of module revalidation.

inputs for the modified module are kept and all other original test cases of T are discarded. If the use of the test cases in T does not satisfy the criterion of the input partition method, which requires at least one test case in each partition class, additional test cases are generated to satisfy the criterion. After the criterion is satisfied, all the newly generated test cases and the original test cases whose execution leads to any modified portion of the module are executed, and the results of the execution are examined. When the existence of errors is detected, debugging of the module will be performed. In the remainder of this section, we will discuss each of these processes in detail.

6.1.1 Derivation Of The Input Partition

The input partition P used in our method is derived by intersecting two input partitions P_s and P_c , which are generated from the program specification and code respectively, and our testing criterion is to have at least one test case in each partition class of P . This input partition has also been considered by Weyuker and Ostrand [WEYU80], who used English for the program specification, and Richardson and Clarke [RICH81], who used a Program Design Language (PDL) type specification. As mentioned before, we used the cause/effect graph to represent the program specification. The partition P_s

can be generated by considering all possible combinations of input conditions from the cause/effect graph and each combination corresponds to a partition class. The partition P_c can be generated by considering that each distinct executable path in a module corresponds to a distinct partition class, except that those paths which differ only in the iteration number of the same loop belong to the same class.

To illustrate the input partition method, let us consider the program which computes the average of a given array of numbers and returns its absolute value. The specification of the program is given in the cause/effect graph shown in Figure 6.2. Its code, together with the program graph information, is shown in Figure 6.3. The cause/effect graph is used to give the input/output relations of the module. Circles on the left correspond to causes, which denote input conditions for the module, and circles on the right correspond to effects, which denote outputs of the module. Circles in the middle denote intermediate nodes, which are used to specify combinations of causes by means of logical relations, such as (AND), (OR) and ~(NOT). Causes (or combinations of causes) and effects are connected if there exist relations: if causes (or combinations of causes) are given in the module, the effects are returned by the module.

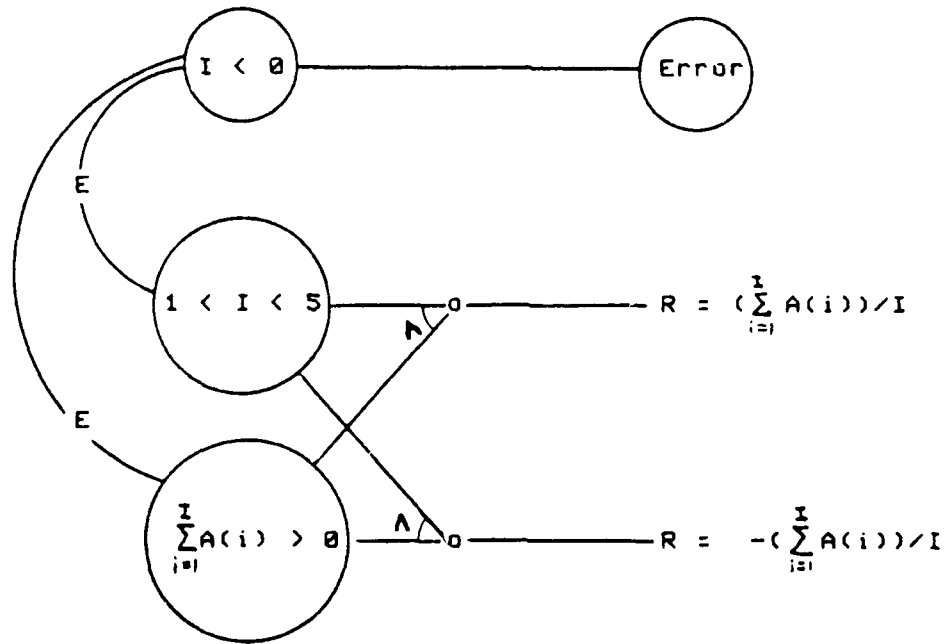


Figure 6.2. The cause/effect graph specification of an example program.

Suppose that the maintenance programmer has corrected two errors: R was not initialized, and "GO TO 40" was "GO TO 50". Let S be $A(i)$. Then the input partition for this program is derived as follows: to generate P_s , we first find the following causes of this example from its cause/effect graph: $I < 0$, $1 < I < 5$ and $S > 0$. However, causes $I < 0$ and $1 < I < 5$ cannot occur at the same time, as indicated in the cause/effect graph by the symbol "E". Similarly, causes $I < 0$ and $S > 0$ cannot occur at the same time. Therefore, the only possible combinations of causes are $I < 0$, $1 < I < 5$ and $S > 0$, and $1 < I$ and $S < 0$, each of which

Program Block No.	Statement No.	SOURCE
1		
	1	SUBROUTINE SUB1 (A,I,R,IERR)
1	1	DIMENSION A(5)
	2	IERR = 0
2	3	IF (I .LE. 0)
	4	\$ GO TO 10
13	4	\$ GO TO 10
	5	IF (I .LE. 1)
3	5	IF (I .LE. 1)
	6	\$ GO TO 20
12	6	\$ GO TO 20
	7	R = 0
4	8	DO 30 L = 1,I
	9	30 R = R + A(L)
5	9	30 R = R + A(L)
	10	R = R/I
6	10	R = R/I
	11	40 IF (R .GT. 0)
7	11	40 IF (R .GT. 0)
	12	\$ GO TO
11	12	\$ GO TO
	13	R = -R
8	13	R = -R
	14	50 WRITE (6,100) R
	15	100 FORMAT (X, F3.2)
9	15	GO TO 60
	16	20 R = A(1)
12	17	GO TO 40
	18	10 IERR = 1
13	18	10 IERR = 1
	19	60 RETURN
10	19	60 RETURN

Figure 6.3. The source code of the example program with the specification shown in Figure 6.2.

represents a partition class of P_s , and $P_s = (I \leq 0) \cup (1 \leq I \leq 5 \text{ and } S > 0) \cup (1 \leq I \leq 5 \text{ and } S \leq 0)$. To generate P_c , we first find the following five different kinds of paths in the program: the path to handle the case $I \leq 0$, the path to handle the case $I=1$ and $S \leq 0$, the path to handle the case that $I=1$ and $S > 0$, the paths to handle the case $2 < I < 5$ and $S \leq 0$, and the paths to handle the case $2 < I < 5$ and $S > 0$. Based on this grouping of paths, P_c can be derived as follows: $P_c = (I \leq 0) \cup (2 < I < 5 \text{ and } S > 0) \cup (2 < I < 5 \text{ and } S \leq 0) \cup (I=1 \text{ and } S > 0) \cup (I=1 \text{ and } S \leq 0)$.

By taking $P = P_s \cap P_c$, we obtain all the partition classes of P as follows:

- class 1. $I \leq 0$
- class 2. $I=1$ and $S > 0$
- class 3. $I=1$ and $S \leq 0$
- class 4. $2 < I < 5$ and $S > 0$
- class 5. $2 < I < 5$ and $S \leq 0$

6.2 Reusability Of Original Test Cases

The changes made to the module may make the application of the original test cases to the modified module invalid. Hence, it is necessary to determine whether the original test cases are correct inputs to the modified module. To do this, we need to examine the total number of input values necessary to invoke

the modified module and the order of these values. For example, because of the modification, another input may be needed to invoke the modified module correctly. In such a case, after the modification, the original test cases are no longer valid to test the modified module and must be discarded. In the case of a modification for error correction, the input which detected the existence of errors should be included in T.

6.3 Assignment Of Original Test Set To The Input Partition Classes

When original test cases are correct inputs to the modified module, we should use them in order to generate fewer new test cases. This is done in our method because it is much easier to see if a given test case satisfies a given input class domain than to generate a new test case which satisfies a given input class domain. As long as some t in T which satisfies the domain constraint of the j th partition class, we assign it to the j th partition class. A similar idea was also used in CASEGEN [RAMA76]. To illustrate this, let us consider a set of original test cases shown in Figure 6.4 for the program shown in Figure 6.3. Since test case 1 satisfies the domain constraint of partition class 1, we assign test 1 to partition class 1. Similarly, test cases 2, 3 and 4 are assigned to partition classes 2, 5 and 4 respectively.

Test case = (I, A(1), A(2), A(3), A(4), A(5))
Test case 1 = (0, 0.0, 0.0, 0.0, 0.0, 0.0)
Test case 2 = (1, 4.3, 0.0, 0.0, 0.0, 0.0)
Test case 3 = (2, -7.89, 2.0, 0.0, 0.0, 0.0)
Test case 4 = (3, 1.58, 6.32, -7.34, 0.0, 0.0)

Figure 6.4. Original test cases prepared for the program in Figure 6.3.

6.4 Selection Of Original Test Cases For Execution

We only need to execute those original test cases which exercise any modified program blocks of the modified module because execution of the rest of the original test cases will only follow the same sequence of the same statements as they did before the module was modified and the same test execution results are generated. Fischer [FISC77] developed a method to select the test cases whose executions exercise the modification, but his method is only applicable to those modifications which do not change the control structure of the program. In this section, we will present a heuristic method, which can be applied to any kind of modifications, including the case where the control structure of the module is changed. Because it is an unsolvable problem to determine which sections

of a module will be traversed for given test cases before their execution, we can determine whether a given test case will traverse any modified portion of the module only during or after its execution. We will first discuss what information is needed to select test cases, and then present the selection algorithm.

6.4.1 Necessary Information For Test Selection

Let us define a path in a program graph as a sequence of nodes and branches. A module path is a path which starts from a node corresponding to the module entry and ends at a node corresponding to a module exit. The reaching set of a node X in the program graph is a set of all possible paths that start from the entry node and end at the node X. The reaching set of a given node in the program graph is identified by using the depth-first search algorithm. We store the reaching set information in the program graph by marking every branch which belongs to some path in the reaching sets of the modified nodes. The reaching set information stored at each branch in the program graph is used to select test cases.

We would use the symbolic execution tree [KING76] to keep track of the test execution information. Each node in the symbolic execution tree corresponds to an execution of a statement. We modify the definition of the symbolic execution

tree so that each node in the tree corresponds to an execution of a Decision-to-Decision Path (DD-path) [HUAN75]. A DD-path is a path in a flow-chart which satisfies the following conditions: 1) its first edge starts either from an entry node or a decision box; 2) its last edge terminates either at a decision box or an exit node; and 3) there are no decision boxes on the path except at both ends. This modification reduces the storage requirements without losing the necessary test selection information. At each node of the modified symbolic execution tree, the information called STATE is stored. STATE is a triple (U, LC, PC) , where U is a vector containing all the values for the variables in the program, LC points to the last statement of the DD-path, and PC stores the constraint to execute the path so far traced.

As the execution proceeds, U , LC , and PC are updated in accordance with the result of the execution. PC is originally assigned to a value of "TRUE" and is updated whenever the execution has gone through a decision point and selected an outcome of it. The new PC is computed by taking an intersection of the old PC and the constraint needed to be satisfied in order to take the selected outcome. Therefore, PC stored at a given node in the symbolic execution tree contains the path constraint for the path between the root node and this node. Note that PC does not change during the execution of a

DD-path. On the other hand, U may change during execution of a DD-path. A value of a variable is changed when a statement assigning some value to this variable is executed. During the execution of a DD-path, possibly existent assignment statements in the DD-path may be executed, changing the values of variables. U stored at a node of the symbolic execution tree contains the values of variables after execution of the sequence of statements corresponding to the path starting from the root node and ending at this node. Note that the U stored at each node as a part of the STATE information is the one computed after the execution of the last statement of the corresponding DD-path.

In addition, we need a table called test information table. This table is needed to keep the test selection information, and it has three columns. The first column is used to store a test case identification, the second column is used to store a symbolic execution tree node identification to show where the test case specified in the first column stopped being executed, and the third column is used to store the information of whether the test case specified in the first column was selected.

6.4.2 Overview Of Selective Test Execution

Our method is developed by utilizing the reaching set information and the symbolic execution tree.

The selective execution using the reaching set of modified nodes can be described as follows: any execution is continued as long as it follows any path in the reaching sets of modified nodes. This is because such a path eventually leads to the execution to a modified node. The execution is terminated as soon as it is found out that the execution does not follow any path in the reaching set of modified nodes. Since all the branches belonging to the paths in the reaching sets of modified nodes are marked, one can tell whether or not the current execution still follows any path in the reaching set of modified nodes by observing if the outcome selected by the execution at each decision point is marked. If the execution so far followed a path in the reaching sets of the modified nodes and a marked branch is selected, we know the execution is still following a path in the reaching sets of the modified nodes. On the other hand, if an unmarked branch is selected, one can tell that the execution no longer follows a path which leads to the execution to a modified node. The execution is continued for the former case while the execution is terminated for the latter case.

The symbolic execution tree is used in order to process more than one test case at one time. The symbolic execution tree is constructed as the data-driven symbolic execution proceeds. Initially, all test cases are stored in Current-Test-Cases (CTC) which holds test cases relevant to the execution. When a decision point of the module is encountered, we must choose an outcome of the predicate because not all the test cases in CTC necessarily evaluate the predicate to the same outcome, and not all outcomes lead to the execution of modified nodes. Based on the selective execution previously described, we have set the outcome selection criterion as follows: When no modified node has been traversed by the execution, select an outcome whose constraint can be satisfied by at least one test case in CTC and whose corresponding branch in the program graph is marked; once one modified node has been traversed, select an outcome whose constraint can be satisfied by at least one test case in CTC.

After the outcome is determined, the test cases in CTC which did not select this outcome are removed and stored in the current tree node. The execution is continued towards the selected outcome of the predicate by adding a new tree node. Note that at that time test cases in CTC are the ones which choose the selected outcome. When the execution is terminated because it has reached an exit point of the module or because

no outcome can be selected, the test selection result is stored in the test information table, using "selected" status for the former and "not selected" for the latter case. Then, the symbolic execution tree is followed backward from the node where the execution was stopped to the root node. When the first tree node containing test cases is found, a new execution is started by assigning the test cases stored in that node to CTC and using the STATE information stored in that node. When no such node is found, the algorithm terminates. Now, let us present the algorithm to select the test cases.

6.4.3 Algorithm To Select Test Cases

Step 1. Set CTC to the original test cases, from which the test cases are selected. Set a counter for the number of traversed modified program blocks, COUNT to 0, the statement pointer ST to the first executable statement, and the tree pointer TN to the root of the symbolic execution tree.

Step 2. If a modified block is traversed, increment COUNT by one. If ST is an exit statement, store the test cases in CTC in the test information table with the status "selected" and go to Step 4. If ST is a decision statement, go to Step 3. Otherwise, set ST to the next statement and repeat Step 2.

Step 3. Select an outcome of a decision statement based on the

outcome selection criterion by using tests in CTC. If no outcome can be selected, all the test cases in CTC are stored in the test information table with the status "not selected" and go to Step 4. Otherwise, store the test cases in CTC which do not satisfy the constraint of the selected outcome in TN and remove these test cases from CTC. Store COUNT in TN. Generate a new tree node as a successor of TN and set TN to this node. Set ST to the next statement and go to Step 2.

Step 4. Trace the tree from TN towards the root. Set TN to the first tree node encountered which holds test cases and go to Step 2. If no such nodes exist, terminate.

6.4.4 An Example

To illustrate the test selection algorithm, let us consider the program shown in Figure 6.3 again. The program graph for this program with the reaching set (program sections 4 and 12 are modified) is shown in Figure 6.5. The addition of $R=0$ is done in program section 4, and the correction of "GO TO 50" to "GO TO 40" is done in program section 12. The original test cases are shown in Figure 6.4. Figure 6.6 (a) and (b) show the symbolic execution tree and the test information table after the execution is over. The tree nodes are numbered as they are generated and attached to the tree. Initially, CTC contains four test cases 1, 2, 3 and 4, and the symbolic

execution tree consists of only the root node.

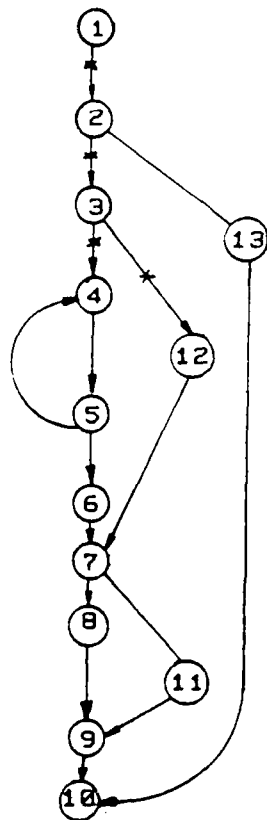
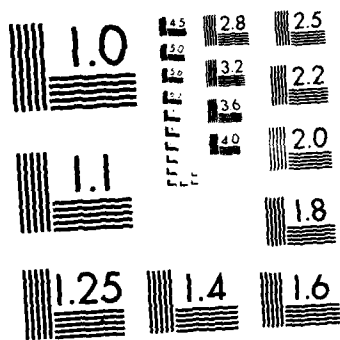
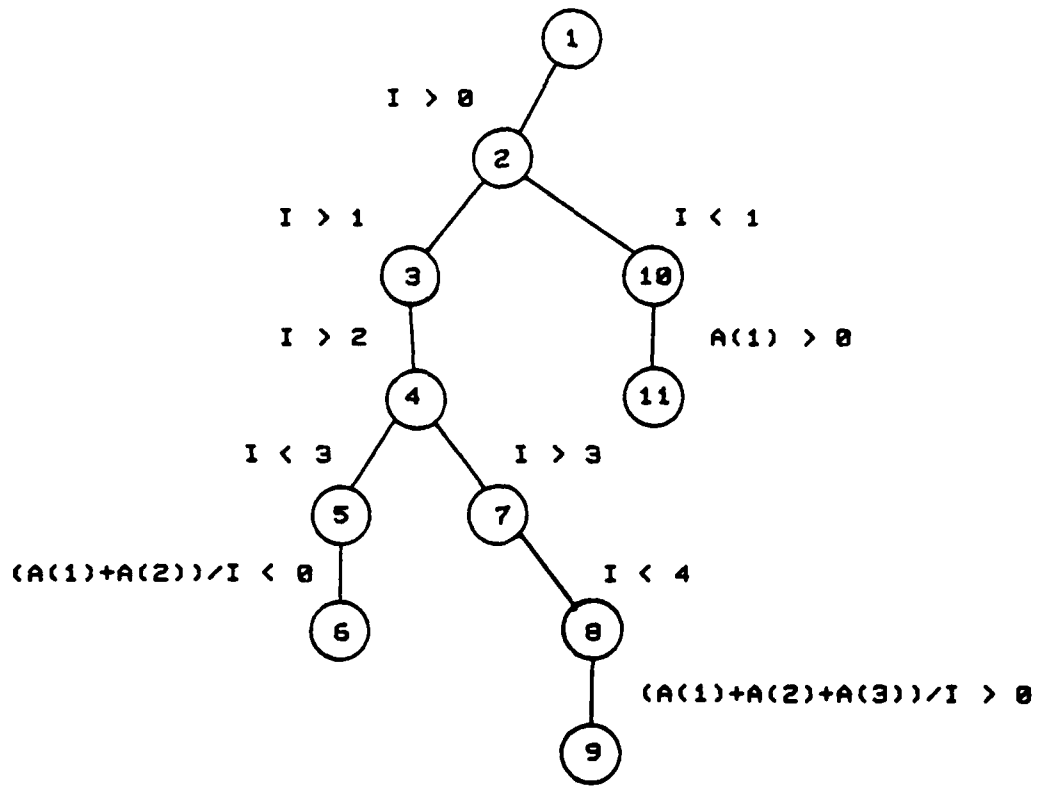


Figure 6.5. The program graph with the reaching set information for the program shown in Figure 6.3.

An execution is started from the first executable statement. When the first decision point, $\neg F(I,LE,0)$, is encountered, we try to select an outcome using the first part of the outcome selection criterion because no modified node has been traversed yet. Since the False outcome of this decision



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



(a)

Test case	Tree node	selection status
3	6	Selected
4	9	Selected
2	11	Selected
1		Not selected

(b)

Figure 6.6. Result of test selection on the program shown in Figure 6.3 with test cases given in Figure 6.4: (a) symbolic execution tree, and (b) contents of test information table.

point corresponds to a marked branch from program section 2 to program section 3 and this outcome can be selected by test cases 2, 3 and 4, the False outcome is chosen. First, test case 1 is removed from CTC because it did not choose the selected outcome and gets stored at the root node. A new tree node is attached to the symbolic execution tree. In the same way, at the next decision point, IF (I.LE.1), the False outcome is selected while test case 2 is stored in the tree node 2. At the next decision point, DO 30 L=1,I, since the modified program section 4 has been already traversed (COUNT = 1), the second part of the outcome selection criterion is used. Since both test cases 3 and 4 select the False outcome, no test case is removed from CTC and the new node 4 is added to the symbolic execution tree. Test cases 3 and 4 do not select the same outcome at the next decision point, DO 30 L=1,I. The True outcome is arbitrarily selected and a new node 5 is attached. Test case 4, which selects the False outcome of this decision point, is removed from CTC and stored at node 4. Hereafter, CTC contains only test case 3. When the current execution terminates at the RETURN statement, the tree is followed backward from node 6 to the root node. The tracing is terminated at node 4 because it is the first node containing a test case, test case 4. A new execution can be started from this node because the STATE information contains the execution

information for the sequence of statements corresponding to a path consisting of tree nodes 1, 2, 3 and 4. Similarly, test cases 4 and 2 are executed. After the execution of test case 2, the tree is followed back to the root node where test case 1 is stored. However, no new execution is started from the root node. The True outcome selected by test case 1 corresponds to a branch between program section 2 and program section 13. This branch is not marked and no modified program section has been traversed (count stored in the root node is zero) and therefore, no outcome is selected. Test case 1 is removed and its status "not selected" is entered in the test information table.

6.5 Test Case Generation And Execution

In order to satisfy the criterion of the input partition method, which requires at least one test case for each partition class, it may be necessary to generate additional test cases. After the assignment of the original test cases is completed, we generate test cases for partition classes which do not have any test cases. Note that when none of the original test cases are reused, we must generate a completely new set of test cases requiring the same amount of effort to test the modified module as a new module. In the example considered in Section 6.3, the partition class 3 is not

assigned any test case. A test case (1, -2.34, 0, 0, 0, 0) is generated and executed to satisfy the testing criterion. The method we have developed has another mode of execution. Under this mode, all the test cases are executed to the end. The algorithm used for this mode can be derived by making a minor modification to the test case selection algorithm.

Algorithm to execute test cases: We made a modification to Step 1 of the Test Selection algorithm. Instead of setting COUNT to 0 initially, it is set to 1. This algorithm executes all the test cases. Since COUNT is always greater than zero, the outcome selection criterion is to select an outcome to which at least one test case in CTC evaluates the encountered predicate. This guarantees that all the test cases are considered and none of them are removed before they reach the exit of the module.

6.6 Output Validation Phase

Our method performs data-driven symbolic execution on the target module and produces symbolic and real outputs, and the domain information. According to Howden [HOWD78] the symbolic outputs are effective to detect computational errors. However, symbolic execution alone is not effective to detect domain errors. We can store the domain information for each partition class in a decision table and this information can be used to

validate the domain information obtained by executing a program. In addition to the domain information, we can add the output information for each partition class in the decision table. The decision table for the example program discussed in Section 6.1 is given in Figure 6.7. Each column of the table corresponds to a partition class, and the upper half of the table is used for storing the domain information and the lower half of the table is used for storing the expected output information.

6.7 Debugging

The method we have developed has a debugging capability called test execution information display. The basic idea of this capability comes from EXDAMS [BALZ69] and ISMS [FAIR75]. Since the necessary test execution information is stored in the symbolic execution tree in the data base, we do not have to execute the program again in order to debug the program. The maintenance programmer can follow a module path forward or backward easily, and retrieve information such as path constraints and the value of each variable in symbolic and real forms from the different locations of the symbolic execution tree.

Partition classes of P	
	1 2 3 4 5
$I < 0$	Y N N N N
$I = 1$	N Y Y N Y
$2 < I < 5$	N N N Y Y
$A(i) > 0$	X Y N N Y
$IERR = 1$	U
$R = A(1)$	U
$R = -A(1)$	U
$R = -A(i)$	U
$R = A(i)$	U

Domain information

Expected outputs information

Figure 6.7. The decision table containing the partition classes derived from specification in Figure 6.2 and the program in Figure 6.3.

Note that the symbolic debugger and the symbolic executor are not the same. Our method has four different kinds of commands which the maintenance programmer can use. The first kind of command, called test specification, is used to specify a test case. The second kind of command, called move command, enables the user to move the pointer within the symbolic execution tree so that the user can retrieve the STATE

information from different tree nodes. The third kind of command, called show command, shows the necessary test execution information that the maintenance programmer requests at different locations of the executed path. The last kind of command, called break point command, can set the break points and stop the tracing at the break points. Since the test execution information is already stored in the data base, additional commands which allow the user to retrieve different debugging information can be easily added without making any modification to the program portions for test execution of the method.

6.8 Discussion And Future Work

Our method employs the input partition method for test generation and data-driven symbolic execution for test execution. The method has been demonstrated by implementing parts of it: 1) selective execution of the original tests, 2) test execution, and 3) debugging. These parts have been implemented in VAX-FORTRAN, using a DEC VAX 11/780 computer under the VAX/VMS operating system, and can be used to analyze programs written in ANSI FORTRAN.

The revalidation of the program after it is modified is very important in the maintenance phase. Presently, no systematic approach exists for revalidating the modified program in the maintenance phase. Our module revalidation technique is developed to assist the maintenance programmer to perform module revalidation for modified modules. We have also developed a set of supporting tools which help the maintenance programmer to apply our revalidation technique. Our module revalidation technique uses the input partition method for test case generation and data-driven symbolic execution for test execution. We only considered programs which can be specified using a cause/effect graph. For this kind of program, it is much easier to derive the input partition from both the program specification and code. The logic of this kind of program is usually straightforward and has no complex loop structures. The cause/effect graph manner of specification was actually used to specify complex real time software systems.

The application of the input partition method tends to produce too many test cases. The number of partition classes should be used as a testability measure, and modularization of the program should be done by taking this factor into consideration in the design stage of the development phase. Although the input partition method requires much effort and time for nontrivial modules, it identifies all the functions of

the module, and in the process of forming the partition, it also detects missing path errors [GOOD75]. Furthermore, it can also detect domain errors. The tool we have developed can select and execute the necessary subset of the original test cases and it can also execute all the generated test cases. The results of the test selection and test execution using data-driven symbolic execution include outputs in symbolic and real forms. The domain information obtained by executing a program can be compared with the correct outputs and the domain information stored in the decision table. This increases the chance of detecting both computational and domain errors. The real value output may detect overflow and truncation errors which cannot be detected by conventional symbolic execution. We used data-driven symbolic execution to solve most of the problems encountered in symbolic execution. When the existence of errors is detected, our tool can be used as a debugger, and provide useful and helpful test execution information for the maintenance programmer. Since module testing is just a part of the overall program revalidation strategy, we plan to develop methods for integration testing and system function testing.

7.8 METRICS RELATED TO SOFTWARE MAINTENANCE

Since the major concern of our research work is with software maintenance problems, we have focused our attention on modifiability related metrics. We have identified several critical attributes that affect modifiability, namely, logical stability, performance stability as well as module strength and coupling. These are all important factors in evaluating the modifiability of a program. Individual measures for each attribute have been developed. There will be a brief description of each of these measures in the following sections. Detailed results have appeared in [YAU78, B0a, B0e, B2c], [EJZAB2]. A limited validation experiment for our logical stability measure has also been conducted. The results will also be presented in the following sections. The integration of these attributes into a modifiability metric requires more study.

Due to the experience gained from the implementation of our logical stability measure, we feel that we need a more efficient way of analyzing logical ripple effects for large scale programs with less requirement on accuracy. In this section, we will also present some preliminary results on these problems.

7.1 Logical Stability Measure

The stability [YAUB0e] of a program is the resistance to the potential ripple effect that the program would have when it is modified. The stability of a module is the resistance to potential ripple effect of a modification of the module on other modules in the program. Since ripple effect is one of the major reasons for introducing errors in the software maintenance process, the stability of a program or module is closely related to its modifiability.

7.1.1 Logical stability measure for modules

A measure for the logical stability of a module k , denoted by LS_k , is defined [YAUB0e] as follows:

$$LS_k = 1/LRE_k$$

where LRE_k = the logical ripple effect measure of a primitive type of modification to a module k , where a primitive type of modification is considered as a modification of a variable definition of module k .

$$= \sum_{i \in U_k} [P(k_i) LCM_{ki}]$$

U_k = the set of all variable definitions in module k ,
 $P(k_i)$ = the probability that a particular variable definition of module k will be selected for

modification,

LCM_{ki} = the logical complexity of each modification to variable definition i in module k

$$= \sum_{t \in W_{ki}} C_t$$

C_t = the complexity measure of module t

W_{ki} = the modules involved in the intermodule change propagation as a consequence of modifying variable definition i of module k

$$= \cup_{j \in Z_{ki}} X_{kj}$$

Z_{ki} = the set of interface variables which are affected by logical ripple effect as a consequence of modification to variable i in module k

X_{kj} = the set of modules involved in intermodule change propagation as a consequence of affecting interface variable j of module k .

Logical stability measure may be normalized to have a range of 0 to 1 with 1 as the optimal logical stability. This normalized logical stability can be utilized qualitatively or it can be correlated with collected data to provide a quantitative measure of stability. The normalized logical stability measure for module k , denoted by LS_k^* , is defined as follows:

$$LS_k^* = 1 - LRE_k^*$$

where LRE_k^* = the normalized logical ripple effect measure for module k

$$= LRE_k^+ / C_p$$

C_p = the total complexity of the program which is equal to the sum of all the module complexities in the program,

LRE_k^+ = the modified logical ripple effect measure for module k

$$= C_k + \sum_{i \in U_k} [P(ki) LCM_{ki}]$$

C_k = the complexity of module k.

7.1.2 Logical stability measure for programs

A measure for the logical stability of a program, denoted by LSP, is defined [YAUB0e] as follows:

$$LSP = 1/LREP$$

where LREP = the measure for the potential logical ripple effect of a primitive modification to a program

$$= \sum_{k=1}^n [P(k) LRE_k]$$

$P(k)$ = the probability that a modification to module k may occur

LRE_k = the logical ripple effect measure of a primitive type of modification to a module k

n = the number of modules in the program.

The normalized logical stability measure for a program, denoted by LSP^* , is defined as follows:

$$LSP^* = 1 - LREP^*$$

where $LREP^*$ = the normalized logical ripple effect measure for the program

$$= \sum_{k=1}^n [P(k) LRE_k^*]$$

$P(k)$ = the probability that a modification to module k may occur

LRE_k^* = the normalized logical ripple effect measure for module k .

7.2 Performance Stability Measure

The performance stability of a module k , denoted by PS_k , is defined as follows:

$$PS_k = 1/PREM_k,$$

where $PREM_k$ is the performance ripple effect measure of a primitive type of modification to a module k and defined as

$$PREM_k = \sum_{i \in U_k} [P(ki) PREB_{ki}]$$

$P(ki)$ is the probability that variable definition i of module k will be modified, U_k is the the set of all variable definitions in module k , and $PREB_{ki}$ is the performance ripple effect of modifying a block i in module k , which is defined as

$PREB_{ki}$ = The number of performance requirements affected by modifying variable i of module k .

The performance stability of a program, denoted by PSP , is defined as follows:

$$PSP = 1/PREP$$

where $PREP$ is the performance ripple effect measure of a primitive type of modification to the program and is defined as

$$PREP = \sum_{k=1}^n [P(k) PREM_k],$$

$P(k)$ is the probability that module k will be modified, and n

is the number of modules in the program.

7.3 Design Stability Measure

It would be more valuable if we can apply the stability measure at early stages of program development. Therefore, we have developed a stability measure that may be applied during the design phase. The design stability of a program, denoted by PDS, is defined [YAUB2c] as follows:

$$PDS = 1/(\sum_x DLRE_x),$$

and the design stability for each module x

$$DS_x = 1/DLRE_x$$

if $DLRE_x \neq 0$, or

$$DS_x = 1$$

if $DLRE_x = 0$, where

$DLRE_x$ = the design logical ripple effect measure for module x

$$= TG_x + \sum_{y \in J_x} TP_{xy} + \sum_{y \in J'_x} TP'_{xy}$$

- TG_x = the total number of assumptions made by other modules about the global data items in GD_x .
- TP_{xy} = the total number of assumptions made by y about the parameters in R_{xy} .
- TP'_{xy} = the total number of assumptions made by y about the parameters in R'_{xy} .
- GD_x = the set of global data defined in module x .
- R_{xy} = the set of passed parameters returned from module x to module y , where $y \in J_x$.
- R'_{xy} = the set of parameters passed from module x to module y , where $y \in J'_x$.
- J_x = the set of modules which invoke module x .
- J'_x = the set of modules invoked by module x .

7.4. Module Strength and Coupling Metrics

We have developed the definitions of metrics for module strength and coupling at the code level, which are presented in detail in [EJZA82]. These metric definitions are approximations of the heuristic definitions of module strength and coupling as found in the literature on Structured Design [MYER78], and are based on a new technique for estimating the probabilities of data object interactions. These metrics are designed to help estimate those qualities of software structure which affect the amount of effort required during the program maintenance activities of functional extension and large-scale modification.

Module strength and coupling appear to be significant attributes affecting the modifiability and reusability of computer programs and should be important elements of future metrics for modifiability and reusability. Metrics for these important structural attributes should also improve the visibility of software structure and provide an objective means for program managers to evaluate individual pieces of software or to choose between alternate solutions to the same problem.

A software tool for computation of the module strength and coupling metrics has been designed for the PASCAL language on our DEC VAX11/780 computer. Implementation of this tool is

nearly complete. Even though the tool is designed for the PASCAL language, our technique is applicable to any block-structured programming language. Validation and refinement of the module strength and coupling metrics should be performed by correlating them to their structured design heuristics in experiments.

In the following sections we will briefly describe our metrics for strength and coupling. The metric algorithms are based on a simple program graph model and estimates of the probabilities of data object interactions. This model characterizes those program attributes most relevant to the metrics.

7.4.1 Estimating Data Object Interaction

Estimates of the probabilities of data object interactions are based on a structural distance function, which assigns an integer value (greater than zero) to each pair of points in program source text (for one procedure or function) where definitions or references to data objects may occur. This function is a count of the number of syntactic levels (associated with statements) in the shortest (syntactic) path from one point to the other. If there is a data flow path from one definition or reference to another, the probability of interaction is assumed to vary inversely with structural

distance. Actual probabilities associated with an average execution path through a program are inaccessible to a static analysis tool.

The structural distance function is used to estimate the probabilities of interaction between any two data object definitions inside a procedure or function in the following four steps:

1) A graph model is created for a procedure or function which consists of edges: a) from nodes where data objects are defined to nodes where they may be referenced, and b) from nodes where data objects are referenced to nodes where data object definitions may be affected. A distance value is assigned to each edge in the graph, which corresponds to an inverse probability of interaction. The distance values are based on the structural distance function described above.

2) The graph model of Step 1 is simplified to indicate only direct distance values between data object definitions. Data object definition nodes correspond to the most important events in an execution path of a program. Each edge in this simplified graph corresponds to a pair of consecutive edges between two data object definition nodes via a reference node. The distance value assigned to each new edge is the sum of the distances associated with the two edges from the original

graph.

3) The transitive closure (or shortest-path) of the matrix of distance values associated with the graph from Step 2 gives the Data Definition Distance Matrix (DDDM) for a procedure. The closure process finds all direct or indirect interactions between data object definition nodes.

4) The DDDM (from Step 3) for a procedure is simplified so that it may be included in the computation of the DDDM's for its calling procedures. Simplification is done by removing all nodes associated with local variables, and by summarizing all data interactions for each parameter and global which is referenced or defined in the procedure with a single input reference and a single output definition. Steps 1-4 are repeated until the DDDM's are constructed for all procedures.

Note that the above steps must be applied to procedures and functions in a specific order so that information is available for a procedure when it is referenced in one of its calling procedures. Any forward referenced procedures in the source text require special iterative processing.

7.4.2 Definition of Intra-Module Strength Metric

We consider a module here as any invocable procedure or function, and define its strength in the context of Structured Design, as the level of interdependence between its subcomponents. We use this definition to construct a strength metric for a procedure or function from its DDDM.

Each element of a DDDM is interpreted as an inverse probability of one data definition node affecting another. Since we consider the data definition nodes to be the most significant nodes in the procedure (in our view of a procedure as a means to alter program data), these nodes are associated with the 'module subcomponents' of the strength definition. The 'level of interdependence' between subcomponents is interpreted as the average probability of interaction between distinct pairs of nodes.

Our strength metric for a procedure A is denoted by $SM(A)$, and is defined as the average over the reciprocals of the elements in the upper triangle minus the main diagonal of the matrix which is the minimum of the $DDDM(A)$ and its transpose.

Two simple examples illustrate the range of values attainable using SM on procedures of clearly different strength: 1) $SM = 1$ for a procedure which initializes the value of its single output parameter, and 2) $SM = 0.2$ for a

procedure which independently initializes 3 output parameters.

7.4.3 Definition of Inter-Module Coupling Metric

Coupling between two modules in a software system is defined, in the context of Structured Design, as the level of direct data object interaction between two modules. We use this definition, along with discussions in the literature about the way in which different situations affect the perception of data coupling, to construct a coupling metric for any two modules in a software system.

There is some form of data object interaction between virtually every pair of modules in a system. We examine only direct coupling between procedures and functions, since the lowest level of coupling is described as having no direct coupling. Direct coupling only occurs between a procedure and its immediate subordinates (those procedures which it may call directly), and between any other procedures which share global data. When the DDDM for a procedure A is computed, all parameter coupling information to its subordinates is available, as well as all global coupling information associated with the global data declared within procedure A.

An Inter-Module Data Object Coupling (IMDOC) value, which is associated with each 'edge' of direct data flow between two procedures A and B will be defined later. One of two different mechanisms, parameter coupling or global coupling, may support each 'edge' of data flow:

Coupling (C) between procedures A and B is defined as follows:

$C(A,B) = \text{Sum of IMDOC}(e) \text{ over each edge } (e) \text{ of direct data flow between A and B.}$

The IMDOC value associated with each edge (e) of direct data flow between any two procedures or functions is defined as follows:

$$\text{IMDOC}(e) = \frac{\text{DOC}(e) \text{ PR}(e) \text{ DR}(e)}{\text{DDDM}(e) \text{ AIP}(e)},$$

where DOC = Data Object Complexity, such as an array is more complex than a simple integer.

PR = Parameter Rating, such as a global variable has a higher value than a parameter.

DR = Data Rating, such as Chapin-type rating for 'through', 'data' or 'control' objects [CHAP79].

AIP = Average Interaction Probability to other elements involved in the coupling.

Now, let us discuss the last four functions DOC, PR, DR and AIP: An increase in data object complexity (DOC) increases the coupling associated with an 'edge' of direct data flow between procedures. This feature is incorporated in the coupling measure in order to take into account of the effects of stamp coupling and in recognition of the fact that a more complex data object has the potential to pass more 'information'. The DOC function is defined recursively according to the structure of the data object.

There is significant experimental evidence to show that direct global coupling renders programs more difficult to modify than direct parameter coupling [DUNSB0]. The parameter rating (PR) function has value 2 for global edges and value 1 for parameter edges.

'Control' objects are understood to contribute more inter-procedural coupling than 'data' objects. 'Through' objects, which are not directly defined or referenced by one (or both) of the procedures involved, but passed through for

use elsewhere, contribute less coupling than 'data' objects. Chapin [CHAP79] defined and discussed 'control', 'data' and 'through' objects and suggested heuristics for identifying each type. An automatable means of roughly identifying objects in this fashion has been defined, and a data rating (DR) value assigned to each type in order to take into account differences in their contribution to coupling. The data rating function has value 2 for 'control' objects, 1/2 for 'through' objects and 1 for 'data' objects.

The grouping of data objects implied by the DDDM probably affects coupling between procedures. If data objects interact closely, they are probably related in function, and contribute less to coupling when they are involved together in direct data flow between procedures, as in the concept of data abstraction. This relationship is made explicit in our measure of coupling with the average interaction probability (AIP) function. AIP is the average of the reciprocals of the distances (interpreted as interaction probabilities) between the source node (v) of the 'edge' under consideration and all other source nodes of edges 1) which contribute to data flow between the same two procedures, 2) which are all either parameters or globals according to v , and 3) whose source nodes are located in the same procedure as v .

7.5 Validation of the logical stability measure

Due to budget constraints, we have performed only a limited number of experiments for validating the logical stability measure at the procedure level. The goal of this validation is to show that there is indeed a certain correlation between the proposed normalized logical stability measure computed for each procedure and the reciprocal of the average number of code changes needed to keep the program consistent and correct caused by a primitive change in that procedure. Therefore, an experiment was devised to quantify the average number of code changes needed for handling the ripple effect caused by actual modifications for procedures in a program. Then, the results were compared with the measures applied to the program.

7.5.1 Experimental Procedures

We will now describe in detail the experiments used to conduct the validation, how programs were selected, how modification proposals were generated for these programs, how the modifications were quantified, how the logical ripple effect of each modification was measured, and the statistical analysis used to determine the correlation figures.

7.5.1.1 Program Selection

A set of programs was prepared for the experiments. These programs were restricted to PASCAL programs because the data flow analysis tools we have developed is for PASCAL programs, although the techniques are applicable to other programming languages. We also limited the use of pointer typed data in the programs because, using existing data flow analysis tools, it would produce imprecise data flow information which will affect the measures generated by the experiments. The length of each of these programs was around 1200 lines of code and each program contained more than 20 procedures. More detailed information about the programs actually selected will be given in Section 7.5.2.

7.5.1.2 Modification Proposal Generation

Specifications for each procedure were generated from the program code and considered for possible modification. Many realistic and feasible modification proposals to these specifications were generated and evaluated for each procedure. In this process, we chose those specifications which were 'local' to a particular procedure as the modification target. Since it was not always possible for all procedures to have meaningful local specifications to be modified, only those

procedures which can satisfy this requirement were selected so that all (or most) of the primary modifications would be within that procedure.

7.5.1.3 Quantification of the Realized Modifications

When the modification proposals were carried out at the code level, three persons were involved in this process. The first person was the author of the target program. All the modifications were performed by the second person, and then were checked for correctness and optimality by the author of the program and the third person. We need to restrict the length of the programs used in the experiments because we want to be sure that every modification could be correctly handled by one person. The number of code changes needed for each modification was quantified as the minimum number of 'tokens' that had to be deleted from or added to the original program in order to implement a particular modification proposal.

7.5.1.4 Actual Ripple Effect Estimation and Normalization

(1) Distinction between primary modifications and the modifications caused by the ripple effect : All modifications made to the procedure, where the specifications to be modified

were generated, were viewed as primary modifications. All other necessary modifications outside the procedure were considered as a result of logical ripple effect.

(2) Normalization : For the i -th modification proposal in procedure M , the above two types of actual resulting modifications were both quantified according to the token-count method. Let the minimum number of tokens involved in the primary modification be P_i , and the minimum number of tokens involved in modification corresponding to the other type be R_i . Then we use $N_i = (R_i + P_i) / P_i$ as the average number of token changes caused by one primitive change (to a token) in the code level of the i -th modification proposal. Suppose we have n modification proposals in procedure M . Since they may vary greatly in the difficulty or efforts involved in making the change, we use their average to estimate the stability of the module. Therefore, the estimated normalized stability measure LS_M^* for procedure M is calculated by

$$LS_M^* = 1 / \left[\left(\sum_{i=1}^n N_i \right) / n \right]$$

This value has a range from 0 to 1, with 1 as the optimal stability which is exactly the same as the proposed stability measure. This result will then be used to correlate with the normalized stability measure calculated from the original program code.

7.5.1.5 Statistical Methods Used in Analysis of the Results

We used Pearson product-moment correlation (r) to analyze our experimental results [BRUN68]. The basic computation formula for the product-moment correlation is

$$r = [N(\sum XY) - (\sum X)(\sum Y)] / [N(\sum X^2) - (\sum X)^2][N(\sum Y^2) - (\sum Y)^2]^{1/2}$$

where N = the number of scores for the pairs (x, y)

$\sum XY$ = the sum of the products of the paired scores

The Pearson product-moment correlation has been widely used to determine if there is a relationship between two sets of paired numbers. The significance of the resulting correlation may be further tested. Two different procedures have been used to test the hypothesis that $r=0$ [BRUN68]. If the sample size N is 30 or larger, a critical-ratio z -test can easily be done. In this case, $z = r(N-1)^{1/2}$ is calculated as an index to find the significance of the correlation. If the sample size N is less than 30, a slightly more complicated t -test should be done. In this case, the degree of freedom df , and the index $t = r[(N-2)/(1-r^2)]^{1/2}$ are calculated to determine the significance of the correlation.

7.5.2 Analysis of the Results

Six programs have been examined in the experimental process of the logical stability measure validation. The average length of the programs used is around 1200 lines of PASCAL code. Each program has between 20 and 47 procedures. The logical stability measure for each procedure in these programs is listed in Table 7.1.

Thirty modification proposals have been generated and applied to 28 procedures. The procedures marked with "*" in Table 7.1 are those which were selected for experimentation. Table 7.2 shows the correlation of logical stability measure versus the experimental result for each modification proposal on those 28 procedures. In order to show that our sampling was representative, the means and standard deviations of the logical stability measures for modules selected in each program have been calculated. As shown in Table 7.2, they are quite close to the means and standard deviations calculated from the logical stability measures of all the modules in individual programs.

The individual correlation and the probability that the hypothesis of the actual correlation being zero is true are both significant and are listed in Table 7.2. The overall correlation coefficient calculated from all the results

(estimated logical stability measure based on our experiment) shown in Table 7.3 against our computed logical stability measure is 0.6338. The probability of the actual correlation being zero is less than 0.1%. These facts indicate that there is indeed a correlation between our computed logical stability measure and the experimental results.

7.5.3 Discussion

The main purpose of this experiment is to show that there is a significant correlation between the proposed normalized stability measure computed for each procedure and the reciprocal of the average number of code changes needed to keep the program consistent and correct after a primitive change has been made to that procedure.

Although the result is positive, refinement of the experimental process should be implemented provided that a better environment and better tools exist.

- 1) The number of code changes is currently calculated by the changes of 'tokens'. When a statement includes a procedure or function call, it should have a suitable weight to reflect the code changes implied by the call.

Table 7.1 Logical stability measures for each module of the target programs used in the experiment.

Program 1:

(A pretty printer for PASCAL program stored in parse-tree form : 1735 lines)

Modulname	Complexity	L.R.E. Factor	Logical stability measure
1 PROGRAM	13	31.94783	0.6091493368
* 2 GETCHAR	8	97.14286	0.0857142806
3 STORENEXTC	2	94.23077	0.1632107496
4 SKIPSPACES	4	93.26087	0.1542533040
* 5 GETCOMMENT	3	60.19512	0.4504771829
* 6 IDTYPE	8	65.00000	0.3652173877
7 GETIDENTIF	7	67.69566	0.3504725695
* 8 GETNUMBER	2	56.26316	0.4933638573
9 GETCHARLIT	4	52.05263	0.5125858188
10 CHARTYPE	7	82.00000	0.2260869741
11 GETSPECIAL	2	58.65116	0.4725986123
12 GETNEXTSYM	7	52.20000	0.4852173924
13 GETSYMBOL	2	59.37879	0.4662714005
14 INITIALIZE	1	90.12000	0.2076521516
15 STACKEMPTY	2	75.00000	0.3304347992
16 STACKFULL	2	7.00000	0.9217391610
17 POPSTACK	2	72.17647	0.3549872637
18 PUSHSTACK	1	73.00000	0.3565217257
19 WRITECRS	3	38.12500	0.6423913240
20 INSERTCR	2	71.00000	0.3652173877
* 21 INSERTBLAN	5	37.09091	0.6339921355
22 LSHIFTON	5	52.30769	0.5016722679
23 LSHIFT	2	60.85714	0.4534161687
24 INSERTSPAC	3	65.92857	0.4006211162
25 MOUVELINEPO	2	24.50000	0.7695652246
* 26 PRINTSYMBO	2	43.53846	0.6040133834
27 PPSYMBOL	5	62.72549	0.4110826850
28 RSHIFTTOCL	2	42.60000	0.6121739149
29 GOBBLE	2	45.94118	0.5831202269
30 RSHIFT	5	57.12000	0.4598261118
** Summary :	115	6.94572	0.4481014609

(Table 7.1 - Continued)

Program 2 :
 (A pretty printer for PASCAL program stored in parse-tree
 form : 1115 lines)

Module name	Complexity	L.R.E. Factor	Logical stability measure
1 PROGRAM	3	56.90909	0.5355884433
2 PF1OLD	0	0.00000	1.0000000000
3 PF1READ	0	0.00000	1.0000000000
4 PF1CLOSE	0	0.00000	1.0000000000
5 MOVE	0	0.00000	1.0000000000
* 6 ADDTOKEN	2	0.00000	0.9844961166
7 SYPARS	4	10.47368	0.8878008723
8 CNSTPT	3	9.81818	0.9006342292
* 9 CNSLST	7	5.35897	0.9041939974
10 VARTYP	2	7.80000	0.9240310192
11 VARLST	5	10.18868	0.8822582960
* 12 TYPTYP	0	4.55844	0.9026477337
13 VARBPT	1	10.71429	0.9091916084
14 TYPEPT	1	10.62500	0.9098837376
15 TYPLST	2	12.27273	0.8893586993
16 BKPARS	1	35.71429	0.7153931260
17 EXPRESSION	21	8.59140	0.7706093192
18 EXPLIST	3	44.68750	0.9303294897
19 VARUSAGE	6	27.35294	0.7414500713
20 ACTUALPARG	4	28.17241	0.7506014705
21 FUNCTIONCA	2	17.30769	0.8503279686
22 CONSTUSAGE	9	0.61818	0.9254404306
23 BEDLST	1	0.00000	0.9922480583
24 STMTLST	10	20.82051	0.7610812783
25 STMPARS	3	49.11111	0.5960379243
26 ASLST	1	41.77778	0.6603893204
27 PSLST	2	18.07692	0.8443649411
28 IFLST	2	47.14286	0.6190476418
29 COLST	5	44.23404	0.6183407903
30 WHLST	1	52.88889	0.5822566748
31 RPLST	1	33.57143	0.7320044041
32 FTLST	2	41.68421	0.6613627076
33 ULST	4	5.05882	0.9297765493
34 PARLST	6	1.47826	0.9420289993
35 BCKLST	7	19.76923	0.7924866080
** Summary :	129	20.65210	0.8215332031

(Table 7.1 - Continued)

Program 3 :
 (A theorem-prover : 1010 lines)

Module name	Complexity	L.R.E. Factor	Logical stability measure
1 PROGRAM	4	1.60000	0.9633986950
2 INITIALIZE	1	0.00000	0.9934640527
3 WRITEARGUM	5	0.00000	0.9673202634
4 WRITELITER	2	3.66667	0.9629629850
5 WRITECLAUS	1	1.90909	0.9809863567
6 JOINNODE	3	136.00000	0.0915032625
7 JOINLITERA	3	136.00000	0.0915032625
8 JOINCLAUSE	3	147.00000	0.0196078420
9 READINCLAU	3	111.27273	0.2531194091
10 READINARG	5	00.16129	0.4433902502
11 READINLITE	2	07.18182	0.4171122909
12 READINSET	4	108.62921	0.2638613582
13 COPYARGUME	3	121.42857	0.1867413521
14 COPYLITERA	2	118.73333	0.2108932734
15 COPYCLAUSE	2	122.92308	0.1835092902
16 COMPAREARG	7	128.11765	0.1168781519
17 COMPARELIT	2	126.85185	0.1578310132
18 COMPARECLA	4	102.85714	0.3015873432
19 REFUTATION	5	113.33334	0.2265794873
* 20 DELETELITE	6	128.41379	0.1214784980
21 CHECKDUPLI	7	118.80000	0.1777777672
22 RESOLVE	3	103.78947	0.3020295581
23 INITIALIZE	2	139.00000	0.0784313679
24 SEPARATEVA	3	136.00000	0.0915032625
25 RESTOREVAR	3	136.00000	0.0915032625
26 ULTVAL	4	135.00000	0.0915032625
27 COLLECT	8	120.00334	0.1628539562
28 COLLECTF	9	100.26144	0.2858729362
29 STARTCOLLE	2	97.85714	0.3473389745
30 MATCH	10	122.49580	0.1340143681
31 UNIFYKEYLI	5	117.92000	0.1966013312
32 APPLYSUBST	8	122.14865	0.1493552327
33 FORMRESOLV	5	109.29358	0.2529831529
34 FINDRESOLV	5	93.00000	0.3542483449
35 SCANRESOLV	8	64.76336	0.5244225264
36 GENERATE	4	26.43478	0.0010798693
** Summary :	153	6.37374	0.3332012892

(Table 7.1 - Continued)

Program 4:

(A time sharing operating system simulator : 1744 lines)

Module name	Complexity	L.R.E. Factor	Logical stability measure
1 PROGRAM	8	115.75000	0.5564516187
2 MTHRANDOM	8	0.00000	1.0000000000
3 CURSOR	1	0.00000	0.9964157939
4 JANRESET	6	204.03847	0.2471739650
* 5 REFRESH	26	0.31923	0.9056658149
* 6 DISPLAY	4	0.54412	0.9837128520
7 FINDLOC	3	230.00000	0.1648745537
8 ERRORCARD	3	55.60000	0.7899641991
9 BINARY	14	217.01587	0.1719861031
10 FINDDATA	5	264.00000	0.0358422995
11 CHECK	4	264.00000	0.0394265056
12 JOBR	11	218.52554	0.1773278117
13 BATCHR	8	147.95062	0.4410371780
14 READCARD	6	245.11111	0.0999601483
* 15 CHECKMAIN	3	138.18182	0.4939719439
16 GENMEM	4	229.00000	0.1648745537
17 GETSID	2	231.00000	0.1648745537
18 GETRID	2	231.00000	0.1648745537
19 GETTIME	1	232.00000	0.1648745537
* 20 ADDSECOND	1	212.89999	0.2333333492
21 REMOVESECO	1	200.16667	0.2789725065
22 ALLOCATECP	3	196.81250	0.2838261724
23 RELEASECPU	1	164.00000	0.4086021781
* 24 ADDRJQ	1	215.05263	0.2256178260
25 ALLOCATEMA	2	198.54839	0.2811886072
26 REMOVERJQ	1	197.27272	0.2893450856
27 RELEASEMAI	2	202.89473	0.2656102777
28 ALLOCATE	6	63.36364	0.7513847947
29 ALLOCATEIO	2	203.39999	0.2637993097
30 RELEASEIOR	1	149.39999	0.4609318972
31 WALLOCATEI	2	203.39999	0.2637993097
32 RELEASEIOW	1	149.39999	0.4609318972
* 33 ADDIORQ	1	212.44444	0.2349661589
34 REMOVEIOQ	3	189.24138	0.3109627962
35 ADDIOWQ	1	212.44444	0.2349661589
* 36 IOINTR	6	100.88889	0.6168857217
37 IOCHECK	10	179.48215	0.3208525181
38 CONVERT	17	73.00000	0.6774193645
39 CONVERTREA	12	90.00000	0.6344085932
40 PUTC	2	74.09091	0.7272727489
41 CLEANRBUF	2	69.00000	0.7455197573
42 PRT	2	51.00000	0.8100358248
43 CSREPORT	35	33.45397	0.7546452880
44 TERMREPORT	13	56.29134	0.7516439557
45 BATCHREPOR	19	75.75000	0.6603942513
* 46 GENCOND	7	187.01819	0.3045942783
47 UPDATE	14	129.15277	0.4869076014
** Summary :	279	10.52915	0.4362154007

(Table 7.1 - Continued)

Program 5:

(An assembler : 823 lines)

Module name	Complexity	L.R.E. Factor	Logical stability measure
1 PROGRAM	1	94.01389	0.1591691375
2 ERROR	1	86.50000	0.2256637216
3 PARSER	13	94.67857	0.0470922589
4 OCTALNO	5	28.00000	0.7079645991
* 5 CHECKMODE	2	87.11111	0.2114060521
6 ENTERDECK	1	9.14286	0.9102401733
7 ENTERSUBNA	4	77.39474	0.2796925902
8 UPDATESUB	1	4.00000	0.9557521939
9 ENTERENTRY	1	9.18182	0.9098954201
10 SEARCHSYM	3	97.00000	0.1150442362
* 11 ENTERSYM	1	99.00000	0.1150442362
12 PRINTSYM	2	1.47059	0.9692868590
13 PRINTDECK	4	1.03175	0.9554712772
14 PASSONE	2	0.00000	0.9823008776
15 CODEGENATR	32	0.00000	0.7168141603
16 NEXTLINE	5	69.67742	0.3391378522
17 FINDIDENT	8	72.94643	0.2836599350
18 PASSTWO	3	11.84210	0.8686539531
* 19 CHECKCODE	2	33.00000	0.6902654767
* 20 NEXTCARD	22	8.46749	0.7303761840
** Summary :	113	12.80153	0.5586465597

(Table 7.1 - Continued)

Program 6:

(A time sharing operating system simulator : 684 lines)

Module name	Complexity	L.R.E. Factor	Logical stability measure
1 PROGRAM	9	57.83133	0.3038403392
2 MTHRANDOM	0	0.00000	1.0000000000
3 ENTERIOQ	1	60.78571	0.3563988209
* 4 IOREAD	1	1.00000	0.9791666865
* 5 IOWRITE	1	1.00000	0.9791666865
6 RELEASEMEM	5	40.97561	0.5210874081
7 FITMEM	11	57.24299	0.2891354561
8 EXCEPTION	18	13.43800	0.6726042032
9 TERMINATE	5	23.39824	0.7042683363
10 ENTERMEMQ	2	83.00000	0.1145833135
11 LEAVEMEMQ	1	66.40000	0.2979166508
12 ENTERCPUQ	1	84.00000	0.1145833135
* 13 ASSIGN	4	0.00000	0.9583333135
14 LEAVECPUQ	2	60.50000	0.3489583135
* 15 SEIZECPU	4	81.00000	0.1145833135
16 COMPETECPU	4	73.65306	0.1911139488
17 LEAVEIOQ	2	83.00000	0.1145833135
18 SEIZEIO	4	79.52728	0.1299242377
* 19 BODY	10	24.25000	0.6432291865
20 CLEANUP	5	66.97222	0.2502893806
21 BATCHREPOR	5	3.60784	0.9103349447
* 22 CARDIN	1	84.00000	0.1145833135
** Summary :	96	8.07312	0.4594855905

Table 7.2. Correlation analysis on logical stability for individual modules.

Program 1:

Module Number	Logical Stability Measure	Experimental Result
2	0.08571	0.43396
5	0.45048	0.40000
6	0.36522	0.33333
8	0.49336	1.00000
21	0.63399	1.00000
26	0.60401	1.00000

**** The correlation coefficient is 0.7221 with
df = 4, t = 2.0878
P (the actual correlation being zero) \leq 10%

**** The mean of the logical stability measures is 0.43880
with standard deviation 0.18193
(The mean for all modules in the program is 0.44810
with standard deviation 0.17988)

Note : The notation P(*) means the probability that * is true.

Program 2:

Module Number	Logical Stability Measure	Experimental Result
6	0.98450	1.00000
9	0.90419	1.00000
12	0.90265	1.00000

**** The correlation coefficient is 0.9966 with
df = 1, t = 12.1477
P (the actual correlation being zero) \leq 5%

**** The mean of the logical stability measures is 0.93045
with standard deviation 0.038227
(The mean for all modules in the program is 0.83010
with standard deviation 0.13609)

Program 3:

Module Number	Logical Stability Measure	Experimental Result
20	0.12148	0.19444

(The mean for all modules in the program is 0.33320
with standard deviation 0.29542)

(Table 7.2 - Continued)

Program 4:

Module Number	Logical Stability Measure	Experimental Result
5	0.90566	0.94186
6	0.98371	0.62162
15	0.49397	1.00000
20	0.23333	0.48971
24	0.22562	0.35210
33	0.23497	0.40749
36	0.61688	1.00000
46	0.30459	1.00000

**** The correlation coefficient is 0.4244 with
df = 6, t = 1.1480
P (the actual correlation being zero) \leq 25%

**** The mean of the logical stability measures is 0.49984
with standard deviation 0.28876
(The mean for all modules in the program is 0.43622
with standard deviation 0.27158)

Program 5:

Module Number	Logical Stability Measure	Experimental Result
5	0.21140	0.35290
11	0.11500	0.45630
19	0.69030	1.00000
20	0.73040	0.54540

**** The correlation coefficient is 0.6866 with
df = 2, t = 1.3354
P (the actual correlation being zero) \leq 30%

**** The mean of the logical stability measures is 0.43678
with standard deviation 0.27605
(The mean for all modules in the program is 0.55865
with standard deviation 0.34267)

Program 6:

Module Number	Logical Stability Measure	Experimental Result
4	0.97917	0.50000
5	0.97917	1.00000
13	0.95833	1.00000
15	0.11458	0.46591
19	0.64323	0.33229
22	0.11458	0.25000

**** The correlation coefficient is 0.6971 with
df = 4, t = 1.9444
P (the actual correlation being zero) \leq 15%

**** The mean of the logical stability measures is 0.63151
with standard deviation 0.38365
(The mean for all modules in the program is 0.45949
with standard deviation 0.32674)

Table 7.3. The summary correlation analysis of logical stability for all modules in the experiment.

Program Number	Logical Stability Measure	Experimental Result
1	0.08571	0.43396
1	0.45048	0.40000
1	0.36522	0.33333
1	0.49330	1.00000
1	0.63399	1.00000
1	0.60401	1.00000
2	0.98450	1.00000
2	0.90419	1.00000
2	0.90265	1.00000
3	0.12148	0.19444
4	0.90566	0.94186
4	0.98371	0.62162
4	0.49397	1.00000
4	0.23333	0.48971
4	0.22562	0.35210
4	0.23497	0.40749
4	0.61688	1.00000
4	0.30459	1.00000
5	0.21140	0.35290
5	0.11500	0.45630
5	0.69030	1.00000
5	0.73040	0.54540
6	0.97917	0.50000
6	0.97917	1.00000
6	0.95833	1.00000
6	0.11458	0.46591
6	0.64323	0.33229
6	0.11458	0.25000

**** The correlation coefficient is 0.6338 with
df = 26, t = 4.1784
P (the actual correlation being zero) < 0.1%

**** The mean of logical stability measures is 0.53859
with standard deviation 0.31947
(The mean for all modules in all programs is 0.50671
with standard deviation 0.30872)

- 2) Due to the limitations of existing data flow analysis tool, we have to limit the use of pointer typed data in the programs to avoid imprecise data flow information. We hope to alleviate this constraint later on.
- 3) The experiments are developed on the procedure level partly due to budget constraints. Experiments on the program level will be more realistic and valuable, but will require more manpower to perform the experiments.
- 4) For large scale programs, a more efficient tool is needed to calculate the proposed measure of the program stability.

7.6. A Unified and Efficient Approach to Logical Ripple Effect Analysis Used in Metrics Calculation

Logical ripple effect analysis is required in computing the logical stability metric for modules and programs [YAU80e]. Theoretically, logical ripple effect analysis has to be performed for each variable occurrence in the program to reveal the logical ripple effect. Therefore, the efficiency of the logical ripple effect analysis technique becomes a prime factor affecting the usability of the metric. The logical ripple effect analysis technique presented in Section 5

emphasizes accuracy in identifying logical ripple effect due to given program modifications rather than efficiency in identifying logical ripple effect for many program modifications for statistical purpose, and hence is not suitable for validating stability measure, especially for large scale programs. Trials using "student projects" or small demonstration experiments are not acceptable representations of the nature of the dynamics encountered in the development of large-scale software systems. Therefore, it is desirable to have an efficient way of performing logical ripple effect analysis.

Software quality metrics are more usable if it can be calculated in the early stages of the program life cycle [KAFUB1]. Strictly code-based metrics provide only an after-the-fact evaluation of the quality of the software structure. Such indications may come too late to correct any structural deficiencies in a program that may already have been completely implemented, possibly at great cost. Typically, it is 100 times more expensive to correct errors in the maintenance phase on large projects than in the requirements phase [BOEHB1]. Therefore, it is also desirable to apply the ripple effect analysis technique during the program design phase, so that data-flow oriented predictive software quality measures may be developed and calculated at that time.

7.6.1 Fomalization of logical ripple effect

We will discuss logical ripple effects caused by only define-preserve-use type data flow propagation as illustrated in Fig. 7.1. That is, in program execution phase, only those ripple effects caused by using data items which were defined somewhere else previously and which may be preserved up to the point where the usage occurs. This is the common understanding and consideration of logical ripple effect in program modification, and it is by no means a severe restriction.

```

A is used to define B
    ::
    ::
    :: There exists a control path
    :: along which B is preserved.
    ::
    ::
    √

```

B is used to define C

Fig. 7.1 An example illustrating that variable A may cause potential logical ripple effect on variable C

Let PP be the collection of all procedures in the program. Let UV be the set of all variable names used in the

program. Without loss of generality, it is assumed that there are no distinct variables of the same name. The scope of a variable may be viewed as an attribute of its name, and the terms "define" and "modify" are used interchangeably.

Now we would like to make the following definitions: DIRECTMOD is defined as a relation from PP to UU such that $(P, V) \in \text{DIRECTMOD}$ implies that V may be directly modified in P . DIRECTUSE is defined as a relation from PP to UU such that $(P, V) \in \text{DIRECTUSE}$ implies that V may be directly used in P . MOD is defined as a relation from PP to UU such that $(P, V) \in \text{MOD}$ implies that V may be modified in P or some subcalls of P . USE is defined as a relation from PP to UU such that $(P, V) \in \text{USE}$ implies that V may be used in P or some subcalls of P . CALL is defined as a relation in PP such that $(P, Q) \in \text{CALL}$ implies that P may call Q directly.

Extending the usual definition of "use-definition chains" to make it fit into inter-procedural data flow analysis, we have the following definitions: For each occurrence of variable v in instruction i of procedure P (denoted by i_p), $\text{DEFS}(v, i_p)$ is defined as the set of instructions which may be the most recent definitions for v at run time. DIRECTMAPTO_P is defined as a relation in UU, where $P \in \text{PP}$ such that $(u, v) \in \text{DIRECTMAPTO}_P$ implies that $(P, u) \in \text{DIRECTUSE}$, $(P, v) \in \text{DIRECTMOD}$ and v is directly modified depending on the

value of u in P . MAPTO_P is defined as a relation in UV , where $P \in PP$. $(u,v) \in \text{MAPTO}_P$ implies that $(P,u) \in \text{USE}$, $(P,v) \in \text{MOD}$ and v is directly modified depending on the value of u in P or some subcalls of P .

The direct logical ripple effect relationship between a pair of variable occurrences is defined as follows: An occurrence of variable u in instruction i_P may impose direct logical ripple effect on an occurrence of variable v in instruction j_Q if and only if $i_P \in \text{DEFS}(u, j_Q)$ and $(u,v) \in \text{DIRECTMAPTO}_Q$. In other words, the pair of variable definitions at two ends of any use-definition chain are said to have direct logical ripple effect from one to the other.

The direct logical ripple effect relationship between a pair of procedures is defined as follows: Procedure P may impose direct logical ripple effect on procedure Q if and only if there exists at least one variable occurrence in P which may impose direct logical ripple effect on a variable occurrence in Q .

DIRECTRIP is defined as a relation in PP , such that $(P,Q) \in \text{DIRECTRIP}$ implies that P may impose direct logical ripple effect on Q .

The logical ripple effect relationship between a pair of

variable occurrences is defined as follows. An occurrence of variable u may impose logical ripple effect on an occurrence of variable v if and only if there exists a sequence of variable occurrences x_1, x_2, \dots, x_n such that

$$u = x_1, v = x_n,$$

and x_i may impose direct ripple effect on x_{i+1} for $1 \leq i \leq n-1$.

The logical ripple effect relationship between a pair of procedures is defined as follows. Procedure P may impose logical ripple effect on procedure Q if and only if there exists at least one variable occurrence in P which may impose logical ripple effect on a variable occurrence in Q .

RIP is defined as a relation in PP such that $(P, Q) \in \text{RIP}$ implies that P may impose logical ripple effect on Q .

7.6.2 Logical ripple effect analysis for metrics calculation

We start with the assumption that no intra-procedural control flow information will be taken into consideration. This is to simulate the situation in the program design phase, where procedures are often viewed as black boxes performing certain functions on interface variables only. Therefore, the algorithm may also be applied in the design phase.

Also, again we stress that the logical ripple effect analysis approach we will present in this section is to emphasize the efficiency consideration in computing logical stability measure. Therefore, the accuracy of the logical ripple effect analysis proposed in this section is somewhat less than that in Section 5. One way of trading accuracy with efficiency is to ignore control flow.

In the following section, for the sake of simplicity, we will not consider mechanisms that may introduce dynamic aliasing among variables, such as reference parameter passing.

7.6.2.1 No Control Flow - No Sharing

From the definitions and the assumption that any intra-procedural execution sequence is possible, we can show that a procedure P may impose direct logical ripple effect on procedure Q if and only if there exists a nonempty set $RIPVAR_{(P,Q)}$ of variables such that $(P,v) \in DIRECTMOD$,

$(Q,v) \in DIRECTUSE$, and $(P,Q) \in (CALL \cup CALL^T \cup ((P,P))^*)^*$ for every $v \in RIPVAR_{(P,Q)}$. Hence, we can compute $DIRECTRIP$ as follows:

$$\text{DIRECTRIP} = (\text{DIRECTMOD DIRECTUSE}^T) \cap (\text{CALL } U \text{ CALL}^T U ((P,P) | P \in PP))^* \quad (7.1)$$

Note that when the call graph is connected, (7.1) may be simplified and become

$$\text{DIRECTRIP} = \text{DIRECTMOD DIRECTUSE}^T \quad (7.2)$$

Then, for each P in PP , we can generate the sets

$$\text{MAP}[P] = \{P_{u,v} \mid (u,v) \in \text{DIRECTMAPTO}_P\} \quad (7.3)$$

$$\text{MAP} = \bigcup_{P \in PP} \text{MAP}[P] \quad (7.4)$$

Now, define RIPPLE1 as a relation in MAP as follows:

$$\text{RIPPLE1} = \{(P_{u,v}, Q_{v,w}) \mid (P,Q) \in \text{DIRECTRIP}, P_{u,v}, Q_{v,w} \in \text{MAP}\} \quad (7.5)$$

This relation is essentially the combination of relation DIRECTRIP with information stored in relation MAPTO .

Finally, we can calculate the relation RIPPLE in MAP by the formula

$$\text{RIPPLE} = \text{RIPPLE1}^* \quad (7.6)$$

It can be shown that the logical ripple effect relation implied by RIPPLE is the most precise information we can have under the assumption that intra-procedural control flow is not considered and no dynamic aliasing condition among variables may occur.

The relation DIRECTRIP can be viewed as the first level inter-procedural ripple effect information while the sets MAPTOCP], for all P_ePP, account for all possible intra-procedural ripple effect information. The way in which RIPPLE is going to be derived in the following sections is analogous to the approach used here. That is, after the relation DIRECTRIP and the sets MAP[P] being established, the relation RIPPLE is calculated according to (7.5) and (7.6). This provides a unified form for this approach which may be applied to both the design and code levels. The derivation of DIRECTRIP and MAPTO[P] may be different depending on various conditions.

7.6.2.2 No Control Flow - Sharing

It has been shown by Barth [BART78] that the possible aliasing relationships among variables caused by call-by-reference parameter passing may be computed by the expression $AFFECT^*(AFFECT^*)^T$, where the relation AFFECT was

defined to be pairs of variables representing the formal-actual reference binding at some point of call. Static aliasing relations can be represented by a set EQU of equivalence classes in a slightly different form such that a pair (u,v) is in EQU if and only if u and v are both in the same equivalence class. EQU can be initialized according to the static aliasing conditions such as REDEFINE in programming language PL/I. Thus, the aliasing relation among variables ALIAS may be computed by

$$\text{ALIAS} = \text{EQU} * \text{AFFECT} * (\text{AFFECT} *)^T. \quad (7.7)$$

Now we can replace (7.1) by

$$\begin{aligned} \text{DIRECTRIP} = & (\text{DIRECTMOD ALIAS DIRECTUSE}^T) \cap \\ & (\text{CALL U CALL}^T \text{U} ((P,P) | P \in PP))^* \end{aligned} \quad (7.8)$$

The correctness of (7.8) can be justified easily. Analogous to (7.2), when the call graph is connected, (7.8) can be simplified and becomes

$$\text{DIRECTRIP} = \text{DIRECTMOD ALIAS DIRECTUSE}^T. \quad (7.9)$$

The precision of (7.9) may be further improved if it is possible to take into consideration the durations of the dynamic aliasing relations implied by $\text{AFFECT} * (\text{AFFECT} *)^T$. (7.3)

thru (7.6) may now be applied to compute the final matrix RIPPLE.

Note that the basic relations, namely DIRECTMOD, DIRECTUSE, DIRECTMAPTO, and CALL, which are needed in the algorithm, are all local information to the procedure and so can be easily constructed from a design document.

The dominant factor in the complexity in computation of this algorithm is the amount of computation of the relation RIPPLE which is the same as the computation of the transitive closure of an array of size |MAP|. The time bound for computing transitive closure of an array of size m is known to be smaller than the order $O(m^3)$. The best result known up to date is of the order $O(m^{2.495364})$ [COPPB1]. This bound depends on the size of the set MAP, which in worse case can be of the order $O(n^2)$, where n is the length of the program. But, this is extremely unlikely in real situations. Actually, based on some empirical data gathered from real programs, $O(n)$ is a more realistic estimate for the size of MAP. Therefore, this gives us an algorithm to compute the total internal ripple effect of any program in a time bound which is independent of the total number of branches in the program. At the same time, since relations can be represented as a bit-matrix, the space bound is manageable too.

7.6.2.3 Control Flow - Tracing

Suppose that we know the way of solving the use-definition chains problem interprocedurally. Then the problem becomes similar to the tracing phase in Section 5, and it may be solved in the following manner: Let the set $RIP1_p$ be

$$RIP1_p = \{ Q \mid \exists v, i_p, j_Q, \text{ s.t. } j_Q \in DEFS(v, i_p) \}.$$

$RIP1$ is in fact a simplified variation of DEFS. With $RIP1$ we can easily build DIRECTRIP as follows

$$\begin{aligned} DIRECTRIP &= \{ (P, Q) \mid \exists v, i_p, j_Q \text{ s.t. } i_p \in DEFS(v, j_Q) \} \\ &= \{ (P, Q) \mid P \in RIP1_Q \}. \end{aligned} \quad (7.10)$$

Sets $MAP[P]$ may be derived by the formula

$$\begin{aligned} MAP[P] &= \{ P_{u,v} \mid (u,v) \in MAPTO_P \} \\ &\quad (DIRECTUSE^T \{ (P,P) \} DIRECTMOD) \end{aligned} \quad (7.11)$$

(7.11) will select local variable pairs from $MAPTO_P$, and (7.4) thru (7.6) may be applied accordingly to yield the relation RIPPLE. It can be shown that the information RIPPLE derived in this manner is precise given that the summary information MOD, USE, MAPTO and $RIP1$ are all precise.

7.6.3 Conclusion

The technique presented here is a somewhat less accurate approach for logical ripple effect analysis than that presented in Section 5.1, but it is more efficient. It is suitable for calculating software metrics because the measure itself is only an estimate of some aspect of the software quality. Precise information is welcome, but sometimes it is too expensive to generate. Approximate information is thus a practical alternative and should not affect much on the quality for validating the metrics. Another advantage of this technique is that it may be applied in both the design and code levels using the same algorithm. This should cause substantial saving in effort on constructing tools for validating the measures and ensures the consistency between measures on different levels.

Although the technique is incomplete in the sense that an efficient way of obtaining the set RIP_1 still needs to be developed, it may be used without considering the control flow within modules which is less precise. Besides the search for an efficient way to generate set RIP_1 , more experiments are needed to give some empirical evidence that the measures calculated in this manner do not differ much from those from the original computation.

7.7 Discussion and Future Work

Metrics for the primary attributes which affect software modifiability are in various stages of development and validation. Metrics for logical stability have been developed and partially validated. Metrics for performance stability, module strength and coupling have been defined. A framework for efficient logical ripple effect analysis approach at both the design and the code levels has been established.

Future work is needed to identify and examine all important software attributes which affect software modifiability and reusability, and to develop a way for combining these attributes into quantitative measures of modifiability and reusability. To achieve this goal, we should develop and validate the metrics related to modifiability, including the metrics for performance stability, complexity, module strength and coupling. Validation and refinement of all related metrics, including modifiability itself, need to be completed by performing a series of comprehensive experiments. Furthermore, significant attributes related to reusability, including portability, need to be identified and examined.

8.0 REFERENCES

- [AH072] Aho, V. A. and Ullman, J. D., The Theory of Parsing, Translation and Compiling, Vol. II, Prentice-Hall, Englewood Cliffs, New Jersey, 1972.
- [ALF077] Alford, M. W., "A Requirements Engineering Methodology for Real-Time Processing Requirements", IEEE Trans. on Software Engineering, Vol. SE-3, No. 1, Jan. 1977, pp. 60-69.
- [ALLE74] Allen, F. E., "Interprocedural Data Flow Analysis", IFIP 74, North-Holland Pub. Co., Amsterdam, 1974, pp. 398-402.
- [ARTH81] Arthur, J. and Ramanathan, J., "Design of Analyzers for Selective Program Analysis", IEEE Trans. on Software Engineering, Vol. SE-7, No. 1, Jan. 1981, pp. 39-51.
- [BALZ69] Balzer, R. M., "EXDAMS - Extendable Debugging and Monitoring System", Proc. AFIPS 1969 Spring Joint Computer Conf., 1969, pp. 567-580.
- [BART78] Barth, J. M., "A Practical Interprocedural Data Flow Analysis Algorithm", Comm. ACM, Vol. 21, No. 9, Sept. 1978, pp. 724-736.
- [BELF77] 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.
- [BOEH73] Boehm, B. W., "Software and Its Impact: A Quantitative Assessment", Datamation, May, 1973, pp. 48-59.
- [BOWL83] Bowles, A. J., Effects of Design Complexity on Software Maintenance, Ph.D. Dissertation, Dept. of Electrical Engineering and Computer Science, Northwestern University, June 1983.
- [BOYD78] Boyd, D. and Pizzarello, A., "Introduction to the WELLMADE Design Methodology", Proc. 3rd Int'l. Conf. on Software Engineering, 1978, pp. 94-100.

- [BRUN68] Bruning, J. L. and Kintz, B. L., Computational Handbook of Statistics, Scott, Foresman and Company, Glenview, IL, 1968
- [CHAP79] Chapin, N., "A Measure of Software Complexity", AFIPS National Computer Conference, pp. 995-1002, Spring 1979.
- [CLAR76] Clarke, L. A., "A System to Generate Test Data and Symbolically Execute Programs", IEEE Trans. on Software Engineering, Vol. SE-2, No. 3, Sept. 1976, pp. 215-222.
- [CLAU79] Claus, U., Ehrig, H. and Rozenberg, G., Graph-Grammars and Their Application to Computer Science and Biology, Lecture Notes in Computer Science 73, Springer-Verlag, 1979.
- [COPP81] Coppersmith, D. and Winograd, S., "On the Asymptotic Complexity of Matrix Multiplication : Extended Summary", Proc. 22nd Annual Sump. on Foundations of Computer Science, IEEE, Oct. 1981, pp. 82-90.
- [DEMA78] DeMarco, T., Structured Analysis and System Specification, Yourdon Inc., 1978.
- [DEME81] Demers, A., Reps, T. and Teitelbaum, T., "Incremental Evaluation for Attribute Grammars with Application to Syntax-directed Editors", Proc. 8th ACM Sump. on Principles of Programming Languages, 1981, pp. 105-116.
- [DUNS80] Dunsmore, H. E. and Gannon, J. D., "Analysis of the Effects of Programming Factors on Programming Effort", Journal of Systems and Software, 1, pp. 141-153, 1980.
- [EJZA82] Ejzak, R. P. Strength and Coupling Metrics of Software Structure, M.S. Thesis, Department of Electrical Engineering and Computer Science, Northwestern University, Evanston, Illinois, August, 1982.
- [FAIR75] Fairley, R. E., "An Experimental Program Testing Facility", IEEE Trans. on Software Engineering, Vol. SE-1, No. 4, Dec. 1975, pp. 350-357.

- [FISC77] Fischer, K. F., "A Test Case Selection Method for the Validation of Software Maintenance Modification", Proc. 1st Int'l. Conf. on Computer Software and Applications (COMPSAC 77), 1977, pp. 421-426.
- [FOSD76] Fosdick, L. D. and Osterweil, L. J., "Data Flow Analysis in Software Reliability", ACM Computing Surveys, Vol. 8, No. 3, Sept. 1976, pp. 305-330.
- [GOOD75] Goodenough, J. B. and Gerhart, S. L., "Towards a Theory of Test Data Selection", IEEE Trans. on Software Engineering, Vol. SE-1, No. 2, June 1975, pp. 156-173.
- [HALL78] Hallin, T. and Hansen, R., "Towards a Better Method of Software Testing", Proc. 2nd Int'l. Computer Software and Applications Conf. (COMPSAC 78), 1978, pp. 153-157.
- [HAY74] Hay, G. G., "Formal Definition of a Simple On-line Teleprocessor in VDL", in Programming Symposium, Paris 1974, Lecture Notes in Computer Science 19, Springer-Verlag, 1974.
- [HECH77] Hecht, M. S., "Flow Analysis of Computer Programs", North-Holland, 1977.
- [HENI79] Heninger, K. L., "Specifying Software Requirements for Complex Systems", Proc. Specifications of Reliable Software, 1979, pp. 1-14.
- [HENI80] Heninger, K. L., "Specifying Software Requirements for Complex Systems: New Techniques and Their Applications", IEEE Trans. on Software Engineering, Vol. SE-6, No. 1, Jan. 1980, pp. 2-12.
- [HOWD78] Howden, W. E., "DISSECT - A Symbolic Evaluation and Program Testing System", IEEE Trans. on Software Engineering, Vol. SE-4, No. 1, Jan. 1978, pp. 70-73.
- [HSIE82] Hsieh, C. C., An Approach to Logical Ripple Effect Analysis for Software Maintenance, Ph. D. Dissertation, Department of Electrical Engineering and Computer Science, Northwestern University, Evanston, Ill., June 1982.
- [HUAN75] Huang, J. C., "An Approach to Program Testing", ACM Computing Surveys, Vol. 7, No. 3, Sept. 1975, pp. 113-128.

- [ICHB79] Ichbiah, J., et al, "Preliminary ADA Reference Manual", ACM SIGPLAN Notices, Vol. 14, No. 6, June 1979, Section 5.2.3.
- [JACK75] Jackson, M. A., Principles of Program Design, Academic Press, 1975.
- [JANS80] Janssens, D. and Rozenberg, G., "Node-Label Controlled Graph Grammars", Proc. 9th Symp. on Mathematical Foundations of Computer Science - Lecture Notes in Computer Science 88, Springer-Verlag, 1980.
- [JENS74] Jensen, K. and Wirth, N., Pascal User Manual and Report, Springer-Verlag, New York, 1974.
- [KAFUB1] Kafura, D. G. and Henry, S. M., "Software Quality Metrics Based on Interconnectivity", Journal of Systems and Software, Vol. 2, No. 2, June 1981, pp. 121-131.
- [KING76] King, J. C., "Symbolic Execution and Program Testing", Comm. ACM, Vol. 19, No. 7, July 1976, pp. 385-394.
- [LEE72] Lee, J. A. N., Computer Semantics, Van Nostrand Reinhold, 1972.
- [LIEN80] Lientz, B. P. and Swanson, E. B., Software Maintenance Management, Addison-Wesley, 1980.
- [LOME77] Lomet, D. B., "Data Flow Analysis in the Presence of Procedure Calls", IBM Journal of Research and Development, Vol. 21, No. 6, Nov. 1977, pp. 559-571.
- [MCCA76] McCabe, T. J., "A Complexity Measure", IEEE Trans. on Software Engineering, Vol. SE-2, No. 6, Dec. 1976, pp. 308-320.
- [MYER76] Myers, G. J., Software Reliability: Principles and Practices, John Wiley and Sons Inc., 1976, pp. 216-246.
- [MYER78] Myers, G. J., Composite/Structured Design, Van Nostrand Reinhold Company, New York, N.Y., 1978
- [PAGAB1] Pagan, F. G., Formal Specification of Programming Languages: A Panoramic Primer, Prentice-Hall, 1981.

- [RAMA76] Ramamoorthy, C. V., Ho, S. F. and Chen, W. T., "On the Automated Generation of Program Test Data", IEEE Trans. on Software Engineering, Vol. SE-2, No. 4 (Dec. 1976), pp. 293-300.
- [RICH81] Richardson, D. and Clarke, L. A., "A Partition Analysis Method to Increase Program Reliability", Proc. 5th Int'l. Conf. on Software Engineering, 1981, pp. 244-253.
- [ROSE79] Rosen, B. K., "Data Flow Analysis for Procedural Languages", Journal of ACM, Vol. 26, No. 2, April 1979, pp. 322-344.
- [ROSS77] Ross, D. T. and Schoman, K. E. Jr., "Structured Analysis for Requirements Definition", IEEE Trans. on Software Engineering, Vol. SE-3, No. 1, Jan. 1977, pp. 6-15.
- [STAY76] Stay, J. F., "HIPO and Integrated Program Design", IBM Systems Journal, Vol. 15, No. 2, 1976, pp. 143-154.
- [SWAN76] Swanson, E. B., "The Dimensions of Maintenance", Proc. 2nd Int'l. Conf. on Software Engineering, 1976, pp. 492-497.
- [TEIT81] Teitelbaum, T., Reps, T. and Horwitz, S., "The Why and Wherefore of the Cornell Program Synthesizer", ACM SIGPLAN Notices, Vol. 16, No. 6, June 1981, pp. 8-16.
- [WASS80] Wasserman, A. I., "Testing and Verification Aspects of Pascal-like Languages", Tutorial on Programming Language Design, IEEE Computer Society Press, 1980, pp. 61-75.
- [WASS82] Wasserman, A. I., "The Future of Programming", Comm. ACM, Vol. 25, No. 3, March 1982, pp. 196-206.
- [WEGN78] Wegner, P., "Research Directions in Software Technology", Proc. 3rd Int'l. Conf. on Software Engineering, 1978, pp. 243-263.
- [WEIS81] Weiser, M., "Program Slicing", Proc. 5th Int'l. Conf. on Software Engineering, 1981, pp. 439-449.

- [WEIS82] Weiser, M., "Programmers Use Slices When Debugging", Comm. ACM, Vol. 25, No. 7, July 1982, pp. 446-452.
- [WEYU80] Weyuker, E. and Ostrand, T., "Theories of Program Testing and the Application of Revealing Subdomains", IEEE Trans. on Software Engineering, Vol. SE-6, No. 3, May 1980, pp. 236-246.
- [WIRT71] Wirth, N., "Program Development by Stepwise Refinement", Comm. ACM, Vol. 14, No. 4, April 1971, pp. 221-227.
- [WIRT75] Wirth, N., "Pascal-S: A Subset and its Implementation", Technical Report 12, Institut fuer Informatik, ETH Zuerich, 1975.
- [YAU78] Yau, S. S., Collofello, J. S. and MacGregor, "Ripple Effect Analysis for Software Maintainance", Proc. 2nd Int'l Conf. on Computer Software and Applications (COMPSAC 78), 1978, pp. 60-65.
- [YAU80a] Yau, S. S., Self-metric Software - Summary of Technical Programs, Final Technical Report RADC-TR-80-138, Vol. I (of 3), NTIS AD-A0386-290, April, 1980.
- [YAU80b] Yau, S. S., Collofello, J. S. and Hsieh, C. C., Self-Metric Software - A Handbook: Part I, Logical Ripple Effect Analysis, Final Technical Report RADC-TR-80-138, Vol II (of 3), NTIS AD-A0386-291, April 1980.
- [YAU80c] Yau, S. S. and Collofello, J. S., Self-Metric Software - A Handbook: Part II, Performance Ripple Effect Analysis, Final Technical Report RADC-TR-80-138, Vol III (of 3), NTIS AD-A0386-292, April 1980.
- [YAU80d] Yau, S. S. and Grabow, P. C., "A Model for Representing the Control Flow and Data Flow of Program Modules", Proc. 4th Int'l. Conf. on Computer Software and Applications (COMPSAC 80), 1980, pp. 153-160.
- [YAU80e] Yau, S. S. and Collofello, J. S., "Some Stability Measures for Software Maintenance", IEEE Trans. on Software Engineering, Vol. SE-6, No. 6, Nov. 1980, pp. 545-552. The Preliminary version of this paper appeared in Proc. 3rd Int'l Conf. on Computer

Software and Applications (COMPSAC 79), 1979, pp.606-611.

- [YAU80f] Yau, S. S. and Collofello, J. S., Performance Ripple Effect Analysis for Large-Scale Software Maintenance, Technical Report RADC-TR-80-55, NTIS AD-A0304-351, March 1980.
- [YAU81a] Yau, S. S. and Grabow, P. C., "A Model for Representing Programs Using Hierarchical Graphs", IEEE Trans on Software Engineering, Vol. SE-7, No. 6, Nov. 1981, pp. 556-574.
- [YAU81b] Yau, S. S., Carvalho, M. B. and Nicholl, R. A., "A Method for Estimating the Execution Time of Arbitrary Paths in Programs", Proc. 5th Int'l Conf. on Computer Software and Applications, (COMPSAC 81), 1981, pp. 225-239.
- [YAU82a] Yau, S. S., Chang, C. K., Hsieh, C.-C., Kishimoto, Z. and Nicholl, R. A., "A Methodology for Software Maintenance", Proc. Int'l. Computer Symposium, Taiwan, 1982, pp. 447-458.
- [YAU82b] Yau, S. S., Grabow, P. C. and Weems, B. P., "A Binary Representation for the Hierarchical Program Model", Proc. 6th Int'l Conf. on Computer Software and Applications, (COMPSAC 82), 1982, pp. 188-195.
- [YAU82c] Yau, S. S. and Collofello, J. S., "Design Stability Measures for Software Maintenance", Proc. 6th Int'l. Conf. on Computer Software and Applications (COMPSAC 82), 1982, pp. 100-100.
- [ZAVE81] Zave, P. and Yeh, R. T., "Executable Requirements for Embedded Systems", Proc. 5th Int'l. Conf. on Software Engineering, 1981, pp. 295-304.
- [ZAVE82] Zave, P., "An Operational Approach to Requirements Specification for Embedded Systems", IEEE Trans on Software Engineering, Vol. SE-8, No. 3, May 1982, pp. 250-269.
- [ZELK78] Zelkowitz, M., "Perspectives on Software Engineering", ACM Computing Surveys, Vol. 10, No. 2, June 1978, pp. 197-216.

9.0 PUBLICATIONS AND PRESENTATIONS

Besides the results of the research presented in this report, many results have already been published or presented in preliminary or complete forms. The publications and presentations are grouped in the following categories: (1) papers, (2) technical reports, (3) presentations related to the project, and (4) Ph.D. dissertations and M.S. theses.

9.1 Papers

1. S. S. Yau and J. S. Collofello, "Some Stability Measures for Software Maintenance", IEEE Trans. on Software Engineering, Vol. SE-6, No. 6, Nov. 1980, pp. 545-552.
2. S. S. Yau, M. B. Carvalho and R. A. Nicholl, "A Method for Estimating the Execution Time of Arbitrary Paths in Programs", Proc. 5th Int'l. Conf. on Computer Software and Applications (COMPSAC 81), 1981, pp. 225-239.
3. S. S. Yau and J. S. Collofello, "Some Design Stability Measures for Software Maintenance", Proc. 6th Int'l. Conf. on Computer Software and Applications (COMPSAC 82), 1982, pp. 100-108.
4. S. S. Yau, C. K. Chang, C.-C. Hsieh, Z. Kishimoto and R. A. Nicholl, "A Methodology for Software Maintenance", Proc. Int'l. Computer Symposium, Taiwan, December 15-17, 1982, pp. 447-458.
5. S. S. Yau and C. C. Hsieh, "Ripple Effect Analysis for Large-Scale Software Maintenance I - Logical Ripple Effect Analysis", submitted for publication.
6. S. S. Yau, J. S. Collofello and R. A. Nicholl, "Ripple Effect Analysis for Large-Scale Software Maintenance II - Performance Ripple Effect Analysis", submitted for publication.

7. S. S. Yau, C. K. Chang and R. A. Nicholl, "An Approach to Incremental Program Modification", submitted for publication.
8. S. S. Yau and Z. Kishimoto, "A Method for Revalidating Programs in the Maintenance Phase - Module Testing", submitted for publication.

9.2 Presentations

1. * S. S. Yau, "Methodologies for Large-Scale Software Maintenance", Seminar, Bell Telephone Laboratories, Naperville, Illinois, July 1, 1980.
2. S. S. Yau, "Performance Stability Measures for Software Maintenance", 3rd Minnowbrook Workshop on Software Performance Evaluation, Blue Mountain Lake, New York, August 19-21, 1980.
3. * S. S. Yau, "Methodologies for Distributed Computing System Software Design", Seminar, Fujitsu Laboratories, Kanagawa-Ken, Japan, October 9, 1980.
4. * S. S. Yau, "Methodologies for Large-Scale Software Maintenance", Seminar, Hitachi Systems Engineering Co., Yokohama, Japan, October 13, 1980.
5. * S. S. Yau, "A Model for Representing the Control Flow and Data Flow of Program Modules", COMPSAC 80, Chicago, Illinois, October 27-31, 1980.
6. * S. S. Yau, "Critical Problem Areas in Software Development", Technical Keynote Speech, Int'l. Computer Symposium 80, Taipei, Taiwan, China, December 16-18, 1980.
7. * S. S. Yau, "Methodologies for Large-Scale Software Maintenance", Seminar, Computer Science Division, Department of Electrical Engineering and Computer Science, University of California at Berkeley, February 25, 1981.
8. S. S. Yau, "A Semantic Program Model for Software Maintenance", 4th Minnowbrook Workshop on Software Performance Evaluation, Blue Mountain Lake, August 11-13, New York, 1981.

9. R. A. Nicholl, "A Method for Estimating the Execution Time of Arbitrary Paths in Programs", COMPSAC 81, Chicago, Illinois, November 18-20, 1981.
10. J. S. Collofello, "Some Design Stability Measures for Software Maintenance", COMPSAC 82, Chicago, Illinois, November 10-12, 1982.
11. *C. K. Chang, "A Methodology for Software Maintenance", Int'l. Computer Symposium, Taiwan, December 15-17, 1982, pp. 447-458.

* These presentations and participation were made at no cost to the contract.

9.3 Technical Reports

S. S. Yau, Methodology for Software Maintenance, RADC Interim Report, July, 1981.

9.4 Dissertation And Theses

A number of graduate students, who have worked on this contract, completed their Ph.D. and M.S degrees in the Department of Electrical Engineering and Computer Science, Northwestern University. Their Ph.D. dissertations and M.S. thesis are listed below:

1. C. C. Hsieh, Logical Ripple Effect Analysis for Program Modification, M.S. Thesis, June, 1980.
2. Z. Kishimoto, Testing for Large-Scale Programs in the Maintenance Phase, Ph.D. Dissertation, June, 1982.

3. C. C. Hsieh, An Approach to Logical Ripple Effect Analysis for Software Maintenance, Ph.D. Dissertation, June, 1982.
4. C. K. Chang, Incremental Modification of Computer Programs, Ph.D. Dissertation, June 1982.
5. R. P. Ejzak, Strength and Coupling Metrics of Software Structures, M.S Thesis, August, 1982.
6. R. S. Wang, Incremental Update of Data Flow Information -- One More Step Toward a Large-Scale Software Maintenance Environment, M.S. Thesis, June, 1983.

10.0 TECHNICAL PERSONNEL

During the period of this study, the following Northwestern University faculty and graduate students contributed to the research effort of this contract:

	<u>1980</u>	<u>1981</u>	<u>1982</u>
<u>Principal Investigator and Project Director</u>	<u>Starting April 23</u>		<u>Ending Nov. 30</u>
Stephen S. Yau	X	X	X
<u>Graduate Students</u>			
Z. Kishimoto	X	X	--
C. C. Hsieh	X	X	--
B. P. Weems	--	X	--
C. K. Chang	X	X	X
R. A. Nicholl	X	X	X
S. C. Chang	--	X	X
R. E. Ejzak	X	X	X
Y. C. Chou	--	X	X
R. S. Wang	--	--	X

In addition, Professor J. S. Collofello of Arizona State University, who worked on the previous project, continued to serve as a consultant to this contract for the work in the areas of software metrics and performance ripple effect analysis. Professor L. Clarke of the University of Massachusetts served as a consultant in the area of testing.

11.8 APPENDIX

For the sake of completeness, we include the following four published papers which contain some of the research results supported by this contract:

1. S. S. Yau and J. S. Collofello, "Some Stability Measures for Software Maintenance", IEEE Trans. on Software Engineering, Vol. SE-6, No. 6, Nov. 1980, pp. 545-552.
2. S. S. Yau, M. B. Carvalho and R. A. Nicholl, "A Method for Estimating the Execution Time of Arbitrary Paths in Programs" Proc. 5th Int'l. Conf. on Computer Software and Applications (COMPSAC 81), 1981, pp. 225-239.
3. S. S. Yau and J. S. Collofello, "Some Design Stability Measures for Software Maintenance", Proc. 6th Int'l. Conf. on Computer Software and Applications (COMPSAC 82), 1982, pp. 100-108.
4. S. S. Yau, C. K. Chang, C. C. Hsieh, Z. Kishimoto and R. A. Nicholl, "A Methodology for Software Maintenance", Proc. Int'l. Computer Symposium, Taiwan, December 15-17, 1982, pp. 447-458.

Some Stability Measures for Software Maintenance

STEPHEN S. YAU, FELLOW, IEEE, AND JAMES S. COLLOFELLO, MEMBER, IEEE

Abstract—Software maintenance is the dominant factor contributing to the high cost of software. In this paper, the software maintenance process and the important software quality attributes that affect the maintenance effort are discussed. One of the most important quality attributes of software maintainability is the stability of a program, which indicates the resistance to the potential ripple effect that the program would have when it is modified. Measures for estimating the stability of a program and the modules of which the program is composed are presented, and an algorithm for computing these stability measures is given. An algorithm for normalizing these measures is also given. Applications of these measures during the maintenance phase are discussed along with an example. An indirect validation of these stability measures is also given. Future research efforts involving application of these measures during the design phase, program restructuring based on these measures, and the development of an overall maintainability measure are also discussed.

Index Terms—Algorithms, applications, logical stability, module stability, maintenance process, normalization, potential ripple effect, program stability, software maintenance, software quality attributes, validation.

I. INTRODUCTION

IT IS well known that the cost of large-scale software systems has become unacceptably high [1], [2]. Much of this excessive software cost can be attributed to the lack of meaningful measures of software. In fact, the definition of software quality is very vague. Since some desired attributes of a program can only be acquired at the expense of other attributes, program quality must be environment dependent. Thus, it is impossible to establish a single figure for software quality. Instead, meaningful attributes which contribute to software quality must be identified. Research results in this area have contributed to the definition of several software quality attributes, such as correctness, flexibility, portability, efficiency, reliability, integrity, testability, and maintainability [3]–[6]. These results are encouraging and provide a reasonably strong basis for the definition of the quality of software.

Since software quality is environment dependent, some attributes may be more desirable than others. One attribute which is almost always desirable except in very limited applications is the *maintainability* of the program. Software maintenance is a very broad activity that includes error corrections,

enhancements of capabilities, deletion of obsolete capabilities, and optimization [7]. The cost of these software maintenance activities has been very high, and it has been estimated ranging from 40 percent [1] to 67 percent [2] of the total cost during the life cycle of large-scale software systems. This very high software maintenance cost suggests that the maintainability of a program is a very critical software quality attribute. Measures are needed to evaluate the maintainability of a program at each phase of its development. These measures must be easily calculated and subject to validation. Techniques must also be developed to restructure the software during each phase of its development in order to improve its maintainability.

In this paper, we will first discuss the software maintenance process and the software quality attributes that affect the maintenance effort. Because accommodating the ripple effect of modifications in a program is normally a large portion of the maintenance effort, especially for not well designed programs [7], we will present some measures for estimating the *stability* of a program, which is the quality attribute indicating the resistance to the potential ripple effect which a program would have when it is modified. Algorithms for computing these stability measures and for normalizing them will be given. Applications of these measures during the maintenance phase along with an example are also presented. Future research efforts involving the application of these measures during the design phase, program restructuring based on these measures, and the development of an overall maintainability measure are also discussed.

II. THE MAINTENANCE PROCESS

As previously discussed, software maintenance is a very broad activity. Once a particular maintenance objective is established, the maintenance personnel must first understand what they are to modify. They must then modify the program to satisfy the maintenance objectives. After modification, they must ensure that the modification does not affect other portions of the program. Finally, they must test the program. These activities can be accomplished in the four phases as shown in Fig. 1.

The first phase consists of analyzing the program in order to understand it. Several attributes such as the complexity of the program, the documentation, and the self-descriptiveness of the program contribute to the ease of understanding the program. The *complexity* of the program is a measure of the effort required to understand the program and is usually based on the control or data flow of the program. The *self-descriptiveness* of the program is a measure of how clear the program is, i.e., how easy it is to read, understand, and use [5].

The second phase consists of generating a particular mainte-

Manuscript received April 1, 1980; revised July 25, 1980. This work was supported by the Rome Air Development Center, U.S. Air Force System Command, under Contracts F30602-76-C0397 and F30602-80-C0139.

S. S. Yau is with the Department of Electrical Engineering and Computer Science, Northwestern University, Evanston, IL 60201.

J. S. Collofello was with the Department of Electrical Engineering and Computer Science, Northwestern University, Evanston, IL 60201. He is now with the Department of Computer Science, Arizona State University, Tempe, AZ 85281.

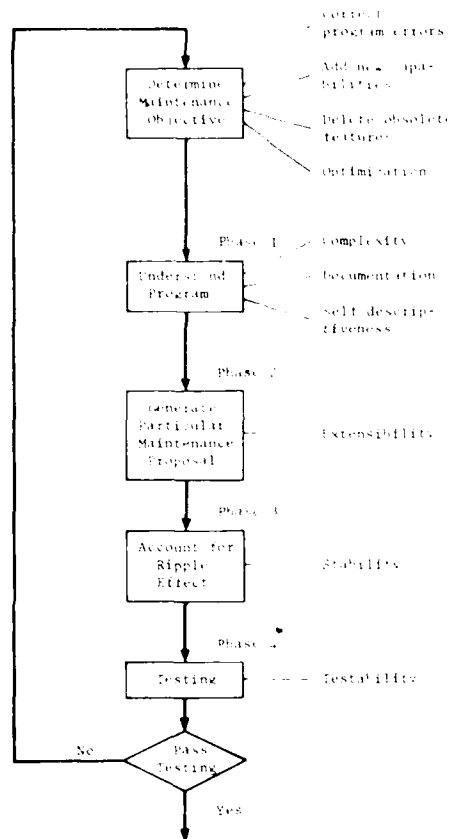


Fig. 1. The software maintenance process.

nance proposal to accomplish the implementation of the maintenance objective. This requires a clear understanding of both the maintenance objective and the program to be modified. However, the ease of generating maintenance proposals for a program is primarily affected by the attribute *extensibility*. The extensibility of the program is a measure of the extent to which the program can support extensions of critical functions [5].

The third phase consists of accounting for all of the ripple effect as a consequence of program modifications. In software, the effect of a modification may not be local to the modification, but may also affect other portions of the program. There is a ripple effect from the location of the modification to the other parts of the programs that are affected by the modification [7]. One aspect of this ripple effect is logical or functional in nature. Another aspect of this ripple effect concerns the performance of the program. Since a large-scale program usually has both functional and performance requirements, it is necessary to understand the potential effect of a program modification from both a logical and a performance point of view [7]. The primary attribute affecting the ripple effect as a consequence of a program modification is the *stability of the program*. Program stability is defined as the resistance to the amplification of changes in the program.

The fourth phase consists of testing the modified program to ensure that the modified program has at least the same reliability level as before. It is important that cost-effective

testing techniques be applied during maintenance. The primary factor contributing to the development of these cost-effective techniques is the *testability* of the program. Program testability is defined as a measure of the effort required to adequately test the program according to some well defined testing criterion.

Each of these four phases and their associated software quality attributes are critical to the maintenance process. All of these software quality attributes must be combined to form a maintainability measure. One of the most important quality attributes is the *stability* of the program. This fact can be illustrated by considering a program which is easy to understand, easy to generate modification proposals for, and easy to test. If the stability of the program is poor, however, the impact of any modification on the program is large. Hence, the maintenance cost will be high and the reliability may also suffer due to the introduction of possible new errors because of the extensive changes that have to be made.

Although the potential benefits of a validated program stability measure are great, very little research has been conducted in this area. Previous stability measures have been developed by Soong [3], Haney [6], and Myers [4]. There exist several weaknesses in these measures which have prevented their wide acceptance. Their largest problem has been the inability to validate the measures due to model inputs that are questionable or difficult to obtain. Other weaknesses of these measures include an assumption that all modifications to a module have the same ripple effect, a *symmetry assumption* that if there exists a nonzero probability of having to change a module i given that module j is changing then there exists a nonzero probability of having to change module j given that module i is changing, and a failure to incorporate a performance component as part of the stability measure.

III. DEVELOPMENT OF LOGICAL STABILITY MEASURES

The *stability* of a program has been defined as the resistance to the potential ripple effect that the program would have when it is modified. Before considering the stability of a program, it is necessary to develop a measure for the stability of a module. The stability of a module can be defined as a measure of the resistance to the potential ripple effect of a modification of the module on other modules in the program. There are two aspects of the stability of a module: the logical aspect and the performance aspect. The *logical stability* of a module is a measure of the resistance to the impact of such a modification on other modules in the program in terms of logical considerations. The *performance stability* of a module is a measure of the resistance to the impact of such a modification on other modules in the program in terms of performance considerations. In this paper, logical stability measures will be developed for a program and the modules of which the program is composed. Performance stability measures are currently under development and the results will be reported in a subsequent paper. Both the logical and the performance stability measures are being developed to overcome the weaknesses of the previous stability measures. In addition, the stability measures are being developed with the following requirements to increase their applicability and acceptance:

- 1) ability to validate the measures,
- 2) consistency with current design methodologies,
- 3) utilization in comparing alternate designs, and
- 4) diagnostic ability.

It should be noted that the stability measures being described are not in themselves indicators of program maintainability. As previously mentioned, program stability is a significant factor contributing to program maintainability. Although the measures being described estimate program stability, they must be utilized in conjunction with the other attributes affecting program maintainability. For example, a single module program of 20 000 statements will possess an excellent program stability since there cannot be any ripple effect among modules, however, the maintainability of the program will probably be quite poor.

Development of a Module Logical Stability Measure

The logical stability of a module is a measure of the resistance to the expected impact of a modification to the module on other modules in the program in terms of logical considerations. Thus, a computation of the logical stability of a module must be based upon some type of analysis of the maintenance activity which will be performed on the module. However, due to the diverse and almost random nature of software maintenance activities, it is virtually meaningless to attempt to predict when the next maintenance activity will occur and what this activity will consist of. Thus, it is impossible to develop a stability measure based upon probabilities of what the maintenance effort will consist of. Instead, the stability measure must be based upon some subset of maintenance activity for which the impact of the modifications can readily be determined. For this purpose, a primitive subset of the maintenance activity is utilized. This consists of a change to a single variable definition in a module. This primitive subset of maintenance activity is utilized because regardless of the complexity of the maintenance activity, it basically consists of modifications to variables in the modules. A logical stability measure can then be computed based upon the impact of these primitive modifications on the program. This logical stability measure will accurately predict the impact of these primitive modifications on the program and, thus, can be utilized to compute the logical stability of the module with respect to the primitive modifications.

Due to the nature of the logical stability of a module, an analysis of the potential logical ripple effect in the program must be conducted. There are two aspects of the logical ripple effect which must be examined. One aspect concerns intramodule change propagation. This involves the flow of program changes within the module as a consequence of the modification. The other aspect concerns intermodule change propagation. This involves the flow of program changes across module boundaries as a consequence of the modification.

Intramodule change propagation is utilized to identify the set Z_k of interface variables which are affected by logical ripple effect as a consequence of a modification to variable definition z in module k . This requires an identification of which variables constitute the module's interfaces and a characterization of the potential intramodule change propa-

gation among the variables in the module. The variables that constitute the module's interfaces consist of its global variables, its output parameters and its variables utilized as input parameters to called modules. Each utilization of a variable as an input parameter to a called module is regarded as a unique interface variable. Thus, if variable x is utilized as an input parameter in two module invocations, then each occurrence of x is regarded as a unique interface variable. Each occurrence must be regarded as a separate interface variable since the complexity of affecting each occurrence of the variable as well as the probability of affecting each occurrence may differ.

Once an interface variable is affected, the flow of program changes may cross module boundaries and affect other modules. Intermodule change propagation is then utilized to compute the set X_{kj} consisting of the set of modules involved in intermodule change propagation as a consequence of affecting interface variable j of module k . In the worst case logical ripple effect analysis, X_{kj} is calculated by first identifying all the modules for which j is an input parameter or global variable. Then, for each of these modules in X_{kj} , the intramodule change propagation emanating from j is traced to the interface variables within the module. Intermodule change propagation is then utilized to identify other modules affected and these are added to X_{kj} . This continues until the ripple effect terminates or no new modules can be added to X_{kj} . An algorithm for performing this worst case ripple effect has already been developed [7], [8].

The worst case ripple effect tracing can significantly be refined if explicit assumptions exist for each module in the program for its input parameters or global variables. Intermodule change propagation tracing would then examine if a module's assumptions have been violated to determine whether it should become a part of the change propagation. If a module's assumptions have not been violated, then the ripple effect will not affect the module.

There are many possible approaches to refining the worst case ripple effect which would not require a complete set of assumptions made for each interface variable for every module. For example, a significant refinement to the worst case change propagation can result by utilizing the simple approach of examining whether or not a module makes any assumptions about the values of its interface variables. These assumptions can be expressed as program assertions. If it does not make any assumptions about the values of its interface variables, then the module cannot be affected by intermodule change propagation. However, if it does make an assumption about the value of an interface variable, then the worst case is automatically in effect and the module is placed in the change propagation resulting from affecting the interface variable if the interface variable is also in the change propagation as a consequence of some modification.

Both intramodule and intermodule change propagation must be utilized to compute the expected impact of a primitive modification to a module on other modules in the program. A measure is needed to evaluate the magnitude of this logical ripple effect which occurs as a consequence of modifying a variable definition. This measure must be associated

with each variable definition in order that the impact of modifying the variable definition during maintenance can be determined. This logical complexity of modification figure will be computed for each variable definition i in every module k and is denoted by LCM_{ki} . There are many possible measures which may be used for LCM_{ki} . All of these measures are dependent upon computation of the modules involved in the intermodule change propagation as a consequence of modifying i . The modules involved in the intermodule change propagation as a consequence of modifying variable definition i of module k can be represented by the set W_{ki} which is constructed as follows:

$$W_{ki} = \bigcup_{l \in Z_{ki}} X_{kl}$$

The simplest measure for LCM_{ki} would be the number of modules involved in the intermodule change propagation as a consequence of modifying i . This measure provides a crude measure of the amount of effort required to analyze the program to ensure that the modification does not introduce any inconsistency into the program. Other measures which examine not only the number of modules involved in the intermodule change propagation, but also the individual complexity of the modules, provide more realistic measures of the amount of effort required to analyze the program to ensure that inconsistencies are not introduced. One such easily computed measure is McCabe's cyclomatic number [9]. The cyclomatic number $V(G)$ is defined in terms of the number of basic paths in the module. A basic path is defined as a path in the module that when taken in combination can generate all possible paths. Computation of the cyclomatic number is, thus, based on a directed-graph representation of the module. For such a graph G_j , the cyclomatic number can be calculated as the number of branches in G_j minus the number of vertices in G_j plus two. Utilizing the cyclomatic number or any other complexity measure, the complexity of modification of variable definition i of module k can be computed as follows:

$$LCM_{ki} = \sum_{l \in W_{ki}} C_l$$

where C_l is the complexity of module l .

Since the logical stability of a module is defined as the resistance to the potential logical ripple effect of a modification to a variable definition i on other modules in the program, the probability that a particular variable definition i of a module k will be selected for modification, denoted by $P(ki)$, must be determined. Now, a basic assumption of utilizing primitive types of maintenance activity is that a modification can occur with equal probability at any point in the module. This implies that each occurrence of each variable definition has an equal probability of being affected by the maintenance activity. Thus, for each module we can calculate the number of variable definitions. If the same variable is defined twice within a module, each definition is regarded separately. The probability that a modification to a module will affect a particular variable definition in the module can then be computed as $1/(\text{number of variable definitions in the module})$.

With the information of LCM_{ki} and $P(ki)$ for each variable definition i of a module k , the potential logical ripple effect

of a primitive type of modification to a module k , denoted by LRE_k , can be computed. The potential logical ripple effect of a module is a measure of the expected impact on the program of a primitive modification to the module. Thus, the potential logical ripple effect can be computed as follows:

$$LRE_k = \sum_{i \in V_k} [P(ki) \cdot LCM_{ki}]$$

where V_k is the set of all variable definitions in module k .

A measure for the logical stability of a module k , denoted by LS_k , can then be established as follows:

$$LS_k = 1/LRE_k$$

Development of a Program Logical Stability Measure

A measure for the potential logical ripple effect of a primitive modification to a program, denoted by $LREP$, can easily be established by considering it as the expected value of LRE_k over all of the modules in the program. Thus, we have

$$LREP = \sum_{k=1}^n [P(k) \cdot LRE_k]$$

where $P(k)$ is the probability that a modification to module k may occur, and n is the number of modules in the program. A basic assumption of utilizing primitive modifications is that a modification can occur with equal probability to any module and at any point in the module. Utilizing this assumption, the probability that a modification will affect a particular module can be computed as $1/n$, where n is the number of modules in the program. This assumption can be relaxed if additional information regarding the program is available. For example, if the program has only recently been released and it is believed that a significant part of the maintenance activity will involve error correction, then the probabilities that particular modules may be affected by a modification may be altered to reflect the probabilities that errors in these modules may be discovered. This can be accomplished by utilizing some complexity or software science measures [10].

A measure for the logical stability of a program, denoted by LSP , can then be established as follows:

$$LSP = 1/LREP$$

IV. ALGORITHM FOR THE COMPUTATION OF THE LOGICAL STABILITY MEASURES

In this section, an algorithm will be outlined for the computation of these logical stability measures. The following description of this algorithm assumes that there does not exist any prior knowledge which might affect the probabilities of program modification, and McCabe's complexity measure [9] is utilized. The algorithm can easily be modified to allow for prior knowledge concerning the probabilities of program modification or to utilize a different complexity measure. The algorithm consists of the following steps:

Step 1. For each module k , identify the set V_k of all variable definitions in module k . Each occurrence of a variable in a variable definition is uniquely identified in V_k . Thus, if the same variable is defined twice within a module, then V_k

contains a unique entry for each definition. The set V_k is created by scanning the source code of module k and adding variables which satisfy any of the following criteria to V_k .

- The variable is defined in an assignment statement.
- The variable is assigned a value which is read as input.
- The variable is an input parameter to module k .
- The variable is an output parameter from a called module.
- The variable is a global variable.

Step 2: For each module k , identify the set T_k of all interface variables in module k . The set T_k is created by scanning the source code of module k and adding variables which satisfy any of the following criteria to T_k .

- The variable is a local variable.
- The variable is an input parameter to a called module.

Each utilization of a variable as an input parameter to a called module is regarded as a unique interface variable. Thus, if variable x is utilized as an input parameter in two module invocations, then each occurrence of x is regarded as a unique interface variable.

- The variable is an output parameter of module k .

Step 3: For each variable definition i in every module k , compute the set Z_{ki} of interface variables in T_k which are affected by a modification to variable definition i of module k by intramodule change propagation [7], [8].

Step 4: For each interface variable j in every module k , compute the set X_{kj} consisting of the modules in intermodule change propagation as a consequence of affecting interface variable j of module k .

Step 5: For each variable definition i in every module k , compute the set W_{ki} consisting of the set of modules involved in intermodule change propagation as a consequence of modifying variable definition i of module k . W_{ki} is formed as follows:

$$W_{ki} = \bigcup_{i \in Z_{ki}} X_{kj}$$

Step 6: For each variable definition i , in every module k , compute LCM_{ki} as follows:

$$LCM_{ki} = \sum_{t \in W_{ki}} C_t$$

where C_t is the McCabe's complexity measure of module t .

Step 7: For each variable definition i in every module k , compute the probability that a particular variable definition i of module k will be selected for modification, denoted by $P(ki)$, as follows:

$$P(ki) = 1/(\text{the number of elements in } V_k).$$

Step 8: For each module k , compute LRE_k and LS_k as follows

$$LRE_k = \sum_{i \in V_k} [P(ki) \cdot LCM_{ki}]$$

$$LS_k = 1/LRE_k$$

Step 9: Compute LREP and LSP as follows:

$$LREP = \sum_{k=1}^n [P(k) \cdot LRE_k]$$

where $P(k) = 1/n$, and n is the number of modules in the program. Then

$$LSP = 1/LREP.$$

V. APPLICATIONS OF THE LOGICAL STABILITY MEASURES

The logical stability measures presented in this paper can be utilized for comparing the stability of alternate versions of a module or a program. The logical stability measures can also be normalized to provide an indication of the amount of effort which will be needed during the maintenance phase to accommodate for inconsistency created by logical ripple effect as a consequence of a modification. Based upon these figures, decisions can be made regarding the logical stability of a program and the modules of which the program is composed. This information can also help maintenance personnel select a particular maintenance proposal among alternatives. For example, if it is determined that a particular maintenance proposal affects modules which have poor stability, then alternative modifications which do not affect these modules should be considered. Modules whose logical stability is too low may also be selected for restructuring in order to improve their logical stability.

The logical stability measures can be normalized by first modifying the computation of the module logical ripple effect measure to include the complexity of the module undergoing maintenance. Let LRE_k^* denote this new logical ripple effect measure for module k which is calculated as follows:

$$LRE_k^* = C_k + \sum_{i \in V_k} [P(ki) \cdot LCM_{ki}]$$

where C_k is the complexity of module k . This enables LRE_k^* to become an expected value for the complexity of a primitive modification to module k . Let C_p be the total complexity of the program which is equal to the sum of all the module complexities in the program. Note that $LRE_k^* \leq C_p$ since the ripple effect is bounded by the number of modules in the program. The normalized logical ripple effect measure for module k , denoted as LRE_k^* , can then be calculated as follows:

$$LRE_k^* = LRE_k^* / C_p.$$

The normalized logical stability measure for module k , denoted as LS_k^* , can then be calculated as follows:

$$LS_k^* = 1 - LRE_k^*.$$

The normalized logical stability measure has a range of 0 to 1 with 1 the optimal logical stability. This normalized logical stability can be utilized qualitatively or it can be correlated with collected data to provide a quantitative measure of stability.

The normalized logical stability measure for the program, denoted as LSP^* , can be computed by first calculating the normalized logical ripple effect measure for the program, denoted as $LREP^*$, as follows:

$$LREP^* = \sum_{k=1}^n [P(k) \cdot LRE_k^*]$$

The normalized logical stability measure for the program can then be calculated as follows:

$$LSP^* = 1 - LREP^*$$

LSP^* has the same range and interpretation as LS_k^* .

VI. EXAMPLE

In this section the logical stability measures for the program in Fig. 2 will be calculated according to the previously described algorithm as follows:

$$LRE_{MAIN} = 4, \quad LRE_{RROOTS} = 2.9, \quad LRE_{TROOTS} = 2.7.$$

The logical stability of each of the modules is given by

$$LS_{MAIN} = 0.25, \quad LS_{RROOTS} = 0.34, \quad LS_{TROOTS} = 0.37.$$

The potential logical ripple effect of the program is

$$LREP = 3.2$$

and hence the logical stability of the program is given by

$$LSP = 0.31.$$

The normalized logical stability measures for each of the modules and the program are given as follows:

$$LS_{MAIN}^* = 0$$

$$LS_{RROOTS}^* = 0.02$$

$$LS_{TROOTS}^* = 0.06$$

$$LSP^* = 0.0267.$$

These measures indicate that the stability of the program in Fig. 2 is extremely poor. An examination of the program provides intuitive support of these measures since the program utilizes common variables in every module as well as shared information in the form of passed parameters. Thus, the change propagation potential is very high in the program.

VII. VALIDATION OF STABILITY MEASURES

As previously mentioned, an important requirement of the stability measures necessary to increase their applicability and acceptance is the capability of validating them. The previous stability measures [3], [4], [6] failed to satisfy this requirement due to calculations involving subjective or difficult to obtain inputs about the program being measured. The stability measures presented in this paper do not suffer from these limitations since they are produced from algorithms which calculate intermodule and intramodule change propagation properties of the program being measured. Thus, these measures easily lend themselves to validation studies.

The stability measures presented in this paper can be validated either directly through experimentation or indirectly through a discussion of how they are influenced by various established attributes of a program which affect its stability during maintenance. The direct approach to validation requires a large database of maintenance information for a significant number of various types of programs in different languages which have undergone a significant number of modifications of a wide variety. One experimental approach would be to examine sets of programs developed to identical

```

C MODULE MAIN
C SUBROUTINE PROGRAM TO COMPUTE STABILITY MEASURES
C AAAAAAAAAA
C COMMON BLOCKS
COMMON /RROOTS/ R1,R2,R3
COMMON /TROOTS/ T1,T2,T3
C READ IN DATA
READ 10, R1,R2,R3
C INITIALIZE STABILITY MEASURES
LS = 0.0
LRE = 0.0
C MAIN PROGRAM
DO 100 CONTINUE
CALL SUBROUTINE RROOTS
WRITE 10, R1,R2,R3
END

C MODULE RROOTS
C SUBROUTINE PROGRAM TO COMPUTE STABILITY MEASURES
C COMMON BLOCKS
COMMON /RROOTS/ R1,R2,R3
C INITIALIZE STABILITY MEASURES
LS = 0.0
LRE = 0.0
C MAIN PROGRAM
DO 100 CONTINUE
CALL SUBROUTINE TROOTS
RETURN
END

C MODULE TROOTS
C SUBROUTINE PROGRAM TO COMPUTE STABILITY MEASURES
C COMMON BLOCKS
COMMON /RROOTS/ R1,R2,R3
COMMON /TROOTS/ T1,T2,T3
C INITIALIZE STABILITY MEASURES
LS = 0.0
LRE = 0.0
C MAIN PROGRAM
DO 100 CONTINUE
RETURN
END

```

Fig. 2. An example program for computing the stability measures.

specifications but differing in design or coding. Logical stability measures for each version of the program could then be calculated to determine which possesses the best stability. A set of identical modifications to the specifications of each program could then be performed. For each modification to each program, a logical complexity of modification, LCM, could then be calculated based upon the difficulty of implementing the particular modification for the program. One particular method for calculating an LCM has previously been described [7], [8]. After a significant number of identical specification modifications have been implemented on all versions of the program, an average logical complexity of modification, ALCM, could be computed for each version of the program. This ALCM reflects the stability of the program and, thus, the ALCM can be utilized as a variable in the experiment. After a significant number of sets of programs have undergone their sets of modifications, experimental conclusions based upon a statistical analysis of the ALCM figures and the stability measures could be formulated.

This direct approach to validation of the stability measures will be difficult due to the number of programs and modifications necessary to produce significant statistical results. Thus, this direct approach to validation will be performed utilizing the maintenance data base which will be created in conjunction with the validation of our program maintainability measure which is currently under investigation.

The stability measures presented here can also be indirectly validated by showing how the measures are affected by some attributes of the program which affect its stability during maintenance. One program attribute which affects maintainability is the use of global variables. The channeling of communication via parameter passing rather than global variables is characteristic of more maintainable programs [11]. Thus,

an indirect validation of the stability measures must show that the stability of programs utilizing parameter passing is generally better than that of programs utilizing global variables. This can be easily shown since the calculation of LS_i is based upon the LCM of each interface variable in module i . Since global variables are regarded as interface variables and since the LCM of an interface variable is equal to the sum of the complexity of the modules affected by modification of the interface variable, LS_i will be small for modules sharing the global variable. Thus, the logical stability of the program will also be small. On the other hand, if communication is via parameter passing instead of global variables, the LCM of the parameters will generally be small, and hence LS_i and LSP will generally be improved. Thus, the stability measures indicate that the stability of programs utilizing parameter passing is generally better than that of programs utilizing global variables.

The stability of a program during maintenance is also affected by the utilization of data abstractions. Data abstractions hide information about data which may undergo modification from the program modules which manipulate it. Thus, data abstraction utilization is characteristic of more maintainable programs. An indirect validation of the stability measures must, therefore, show that the stability of programs utilizing data abstractions is generally better than that of programs whose modules directly manipulate data structures. This can easily be shown by examining the stability measures of a program that utilizes data abstractions and comparing those measures to that of an equivalent program in which the modules directly access the data structure, i.e., data abstractions are not utilized. The modules which utilize a data abstraction to access a data structure will have fewer assumptions about their interface variables and hence have higher stability than that of the modules directly accessing the data structure and hence having many assumptions about it. For example, consider a data structure consisting of records where each record has an employee number and a department number. Assume that module INIT initializes the data structure and orders the records by the employee number. Also, assume modules X, Y, and Z must access the data structure to obtain the department for a given employee number. In this design, if module INIT is modified so that the records in the data structure are ordered by the department instead of the employee number, then modules X, Y, and Z must also be modified. This potential modification is reflected in the calculation of LS_{INIT} and, consequently, LSP . If, however, modules X, Y, and Z access the data structure through a data abstraction, then the same modification to module INIT will affect the data abstraction algorithm, but not modules X, Y, and Z. Consequently, LS_{INIT} and, consequently, LSP will be larger in the program which utilizes the data abstraction than the measure for the program which does not. Thus, the stability measures proposed in this paper indicate that the stability of programs utilizing data abstraction is generally better than that of programs which do not.

Another attribute affecting program stability during maintenance is a program's control and data structure in which the scope of effect of a modification lies within the scope of control of the modules. This implies that the only part of a program

affected by a change to a module, i.e., its scope of effect, is a subset of the modules which are directly or indirectly invoked by the modified module, i.e., its scope of control [12]. An indirect validation of the stability measures must, therefore, show that the stability of programs possessing this type of control and data structure are better than that of programs which do not possess this attribute. Now a program which exhibits this scope of effect/scope of control property has a logical stability which is calculated from the logical stability of its modules, each of which is bounded above by the sum of the complexity of the modules which lie within its scope of control. If the scope of effect of a modification to a module does not lie within the scope of control of the module, the logical stability of the module is only bounded above by the complexity of the entire program. Thus, the stability measures indicate that the stability of programs possessing the scope of effect/scope of control attribute are generally better than that of programs which do not possess this attribute.

Another attribute affecting program stability during maintenance is the complexity of the program. Program complexity directly affects the understandability of the program and, consequently, its maintainability. Thus, an indirect validation of the stability measures must, therefore, show that the stability of programs with less complexity is generally better than that of programs with more complexity. This is readily apparent from the calculation of the logical complexity of modification of an interface variable. Thus, complexity is clearly reflected in the calculation of the stability measures.

The stability measures presented here can, thus, be indirectly validated since they incorporate and reflect some aspects of program design generally recognized as contributing to the development of program stability during maintenance.

VIII. CONCLUSION AND FUTURE RESEARCH

In this paper, measures for estimating the logical stability of a program and the modules of which the program is composed have been presented. Algorithms for computing these stability measures and for normalizing them have also been given. Applications and interpretations of these stability measures as well as an indirect validation of the measures have been presented.

Much research remains to be done in this area. One area of future research involves the application of the logical stability measures to the design phase of the software life cycle. An analysis of the control flow and the data flow of the design of the program should provide sufficient information for calculation of a logical stability measure during the design phase.

Another area of future research involves the development of a performance stability measure. Since a program modification may result in both a logical and a performance ripple effect, a measure for the performance stability of a program and the modules of which the program is composed is also necessary [7], [8].

Much research also remains to be done in the identification of the other software quality factors contributing to maintainability. Suitable measures for these software quality factors must also be developed. These measures must then be integrated with the stability measures to produce a maintainability

measure. This maintainability measure must be calculatable at each phase of the software life cycle and must be validated.

Another area of future research involves the development of automated restructuring techniques to improve both the stability of a program and the modules of which the program is composed. These restructuring techniques should be applicable at each phase of the software development. Restructuring techniques must also be developed to improve the other quality factors contributing to maintainability. These restructuring techniques must automatically improve the maintainability of the program at each phase of its development. The net results of this approach should be a significant reduction of the maintenance costs of software programs and, consequently, a substantial reduction in their life cycle costs. Program reliability should also be improved because fewer errors may be injected into the program during program changes due to its improved maintainability.

REFERENCES

- [1] B. W. Boehm, "Software and its impact: A quantitative assessment," *Datamation*, pp. 48-59, May 1973.
- [2] M. V. Zelkowitz, "Perspectives on software engineering," *ACM Comput. Surveys*, vol. 10, pp. 197-216, June 1978.
- [3] N. L. Soong, "A program stability measure," in *Proc. 1977 Annu. ACM Conf.*, pp. 163-173.
- [4] G. J. Myers, *Reliable Software through Composite Design*. Petroselli Charter, 1979, pp. 137-149.
- [5] J. A. McCall, P. K. Richards, and G. T. Walters, *Factors in Software Quality, Volume III: Preliminary Handbook on Software Quality for an Acquisition Manager*, NTIS AD-A049 055, Nov. 1977, pp. 2-1-3-7.
- [6] E. M. Haney, "Module connection analysis," in *Proc. AFIPS 1972 Fall Joint Comput. Conf.*, vol. 41, part I, pp. 173-179.
- [7] S. S. Yau, J. S. Collofello, and T. M. MacGregor, "Ripple effect analysis of software maintenance," in *Proc. COMPSAC '78*, pp. 60-65.
- [8] S. S. Yau, "Self-metric software: Summary of technical progress," Rep. NTIS AD-A086-290, Apr. 1980.
- [9] T. J. McCabe, "A complexity measure," *IEEE Trans. Software Eng.*, vol. SE-2, pp. 308-320, Dec. 1976.
- [10] M. H. Halstead, *Elements of Software Science*. New York: Elsevier North-Holland, 1977, pp. 84-91.
- [11] L. A. Belady and M. M. Lehman, "The characteristics of large systems," in *Research Directions in Software Technology*, P. Wegner, Ed. Cambridge, MA: MIT Press, 1979, pp. 106-139.
- [12] L. Yourdon and L. Constantine, *Structured Design*. Yourdon, 1976.

Stephen S. Yau (S'60-M'61-SM'68-1773), for a photograph and biography, see p. 434 of the September 1980 issue of this *TRANSACTIONS*.



James S. Collofello (S'78-M'79) received the B.S. and M.S. degrees in mathematics/computer science from Northern Illinois University, DeKalb, in 1976 and 1977, respectively, and the Ph.D. degree in computer science from Northwestern University, Evanston, IL, in 1978.

After graduating, he was a visiting Assistant Professor in the Department of Electrical Engineering and Computer Science, Northwestern University. He joined the faculty of the Department of Computer Science, Arizona State University, Tempe, in August 1979 and is currently an Assistant Professor there. He is interested in the reliability and maintainability of computing systems and the development, validation, and application of software quality metrics.

Dr. Collofello is a member of the Association for Computing Machinery and Sigma Xi.

A METHOD FOR ESTIMATING THE EXECUTION TIME OF ARBITRARY PATHS IN PROGRAMS*

S.S. Yau, M.B. Carvalho and R.A. Nicholl,
 Department of Electrical Engineering and Computer Science,
 Northwestern University, Evanston, Illinois 60201

One measure of program performance is the execution time of the program. In this paper a technique based on a self-metric approach for estimating the execution time of program paths is presented. Estimates are obtained for each of the operations of a programming language. A program is then used to analyze the program to be measured, inserting additional program instructions to obtain statistics regarding the execution time. This work was being done to assist in the analysis of performance ripple effect during program modification. In this application, information may be needed about each execution of specific paths with critical timing constraints. The particular paths to be measured, and the type of statistics to be provided are determined by the user.

This technique has been implemented and used for experiments with PASCAL programs running on a DEC VAX 11/780 computer.

Index Terms -- Dynamic monitoring, execution time, hardware clocks, PASCAL programming language, performance ripple effect, program performance.

INTRODUCTION

Program performance is a measure of how efficiently a sequence of statements of a computer program is executed in a given environment. Ideally, one should be able to determine an absolute figure which would be a measure of the performance of a program and remain invariant, regardless of the environmental conditions. However, as will be shown later, such a measure is extremely difficult to define, and hence the restriction to "performance in a given environment" is made.

A number of program performance indices have been proposed, and the one that is most widely accepted is that of

execution time [1]. It is assumed, for the time being, that an intuitive definition of execution time is "the amount of time required to execute a portion of code". A more detailed discussion and definition of execution time will be given later.

In this paper, we will present a technique to provide an estimate of the execution time of any set of statements in a program. The reason behind the effort to provide such information is threefold:

1. to provide the user with assistance in the decision making process about program efficiency when selecting different algorithms,

2. to provide a tool for the development of faster and more reliable programs. By having a frequency count of different modules of a program, the user will be able to recognize areas of code which are never executed (indicating redundancy or possible error), and also those areas on which to concentrate optimizing efforts (the performance bottlenecks and heavily used procedures). It has been reported that, for a typical program, approximately 3% of the code accounts for 50% of the execution time [2],

3. to provide the software maintenance personnel with an easy-to-use tool to detect and measure the performance ripple effect [3,4]. Performance ripple effect has been defined as the change in the performance of modules as a consequence of software modifications, and is due to the existence of a performance dependency relationship between two modules, say A and B; that is, a change in module A can have an effect on the performance of module B [3,4]. Consider two modules, A and B, from a given program, as shown in Figure 1. Module B can be affected by a change in module A if there is any kind of linkage between A and B, such as a control and/or data flow link. A change in statement S in A may affect the execution time - a performance index - of B. In order to check the analysis of performance ripple effects and to test

* This work was supported by the Rome Air Development Center, U.S. Air Force Systems Command, under Contract #30602-80-C-0139.

the correctness of a modification, it is necessary to test the performance of all paths in the program which have performance requirements.

The technique we are going to present is based on self-metric analysis of program performance. We describe a set of pseudo-statements to be inserted by the programmer into the program to be analyzed. This is the method by which the critical paths of the program are defined. In order to estimate the execution time of a path, we must know the time required for each operation in the high-level language. Experiments are performed to determine the average execution time of each operation in the high-level language. We refer to these values as the costs of the operations. Using a table of costs for the operations of the language and the pseudo-statements inserted by the programmer, a source language program analyzer modifies the source program to include additional statements to update cost counters associated with the paths being timed. To demonstrate the technique, an analyzer has been implemented for PASCAL programs, using a cost table for a DEC VAX 11/780 computer. We will compare the results obtained for two programs under various conditions, using both the system clock and the analyzer.

DEFINITION OF EXECUTION TIME

A desirable characteristic of any measure is that it must be repeatable, otherwise it would be of little use, if any. This condition restricts the definition of execution time since the execution time of a program may mean one of several different things, depending on the point of view from which it is being considered.

In a uniprogramming environment, measuring CPU time is an easy task that requires only access to a real-time clock which can keep time for any desired time unit. CPU time charged to a process, or time units used for the execution of the instructions between two points in a program are due only to the execution of that particular segment.

A user in a multiprogramming environment may associate execution time with turnaround time; that is, the span of time from the moment the execution command is issued until the moment the task is completed; alternatively, the execution time of the program may be viewed as a measure of the program's consumption of (virtual) CPU time - real time minus interrupts - or it may signify CPU time plus the execution of all necessary system routines.

Thus it is clear that there are several sources of variation associated

with CPU time consumed, roughly classifiable into two broad categories: variations in hardware speed and effects of system software. The former includes mixed memory speeds for the different levels of memory hierarchy, cache performance, and the size of the allocated working set. Software factors include the cost of processing interrupts ("quick" interrupt service routines are sometimes charged to whichever process was interrupted, because it would not be worth the effort to charge it to the appropriate process, context switching or supervisor/monitor services) and the cost of scheduling and statistical work.

Obviously, any arrangements to reduce these effects would restrict the utilization of the system by other users, and make any measuring session cumbersome and exceedingly complex.

It is true that, today, most operating systems do keep track of the CPU time used by executing user processes. But, besides the fact that such information is plagued by the variations already discussed, they still retain the most problematic aspect of timing procedures and measuring CPU usage, namely clock resolution [5,6]. The clock resolution should be small when compared with the time spent in the procedure. But unfortunately, this is not true of most systems.

The IBM/370 hardware includes a time-of-day clock (real time clock) with a resolution of 1 microsecond, which runs continuously and provides timing information for operating system scheduling and accounting purposes. The clock is easily accessible with one low-level instruction (move register type), and has been successfully used to time procedures [7].

The DEC VAX-11/780 architecture, on the other hand, presents a very restrictive clock system, as shown in Figure 2. The CPU time information stored in the process header can be read by means of a system routine available in the VAX/VMS Operating System: the "Get Job/Process Information" system service provides accounting, status and identification information about a specified process [8]. The accumulated CPU time may be read in 10 millisecond "tics".

Therefore, when trying to measure the actual execution time of procedures by means of the virtual CPU time, one runs into two levels of difficulty: the first level is the problem of accessing the operating system clock registers, and their inadequate (too coarse) accuracy; the second level is the variation associated with the virtual CPU time due to memory management, interrupts, operating system service routines and

shortcuts in the accounting policy, and the overhead introduced by the use of software probes that call system routines.

In view of all these considerations, we shall define execution time as the amount of CPU time used by a program when a sequence of statements is executed, regardless of the environmental conditions. The execution time of a sequence of statements, then, is the sum of the execution times of the statements of the sequence, each of which is, in turn, the sum of the execution times of the operations performed within those statements.

METHODS OF MONITORING PROGRAM EXECUTION

There are a number of different approaches used to monitor the behavior of a program. Lyon and Stillman [9], list four typical monitoring philosophies and compare them on the basis of a number of characteristics of cost and convenience, such as portability, accuracy, cost to prepare a program, and clock requirements.

Each of the four methods has its advantages and disadvantages. The first method, using clock interrupts via an operating system, is excellent for use across different compilers, but requires a fast clock for good accuracy and precision. The second method, event-driven hardware probes, are good in every respect, but are costly to set up. The third method, inserting calls to a system clock limits use to one language, but is an excellent approach if there is a consistent clock, and if the operating system keeps track of program-state and supervisor-state times separately and for each user. The fourth method, placing counters inside a segmented program, although limited to one language and requiring knowledge of the approximate cost for each statement type, does have a great advantage: during execution of the program, performance monitoring does not use any clock.

This latter method is also known as the "self-metric" approach, due to the fact that an instrumented version of the program gathers all the information about itself, in addition to performing its normal function. This is a method which has had a number of reported uses, for monitoring both the execution time and the logical behavior of the program [2,9-15].

The self-metric approach typically consists of two phases: the original source code is first accepted as input by a source code analyzer, which produces as output the instrumented version of the program, containing the necessary code for the tallying function. Phase two

occurs when the augmented version of the program is actually executed on the user's original input data, producing a report on the execution statistics in addition to its normal output. The entire process is represented in Figure 3.

THE PSEUDO-STATEMENTS FOR INSTRUMENTATION PURPOSES

Four pseudo-statements and one pseudo-declaration are defined to allow the user to instrument the source code, issuing directives to the analyzer to take actions such as to set up a new counter, to turn a counter on or off (thus defining the segment of code that is to be monitored), to reset the value of the counters, and to prepare the data file for output and record the results.

The VAR Pseudo-declaration

This pseudo-declaration is inserted by the user in the global variable declaration part of the source program. It instructs the analyzer to generate the code necessary to declare the global variables which will keep track of the statistical measures gathered during the execution of the instrumented version. The names of the probes are declared in the program by the user. The syntax of this pseudo-declaration is:

```
$$ VAR [C-1 , ]* C-n ,
```

where C-1 ... C-n are the names of the probes. These names must satisfy the syntax for an identifier of the programming language in use.

The INIT Pseudo-statement

The INIT pseudo-statement initializes all variables used for tallying purposes, and opens and prepares for output the data file into which the measurements will be written. This pseudo-statement should be used as the first statement of the main program. The syntax of this pseudo-statement is:

```
$$ INIT
```

The ON Pseudo-statement

This pseudo-statement opens the scope of a new probe and defines the starting point of a new segment of code which is to be time-monitored. The syntax of this pseudo-statement is:

```
$$ ON PROBENAME
```

where PROBENAME is one of the probe names which were declared in the VAR pseudo-declaration, which has not yet been used.

The OFF Pseudo-statement

The OFF pseudo-statement closes the scope of the current probe, reestablishes the scope of the previous probe (there is one predefined probe), and generates source code, which will prepare collected data and do simplification and output of intermediate results. The user may specify whether or not he/she wants every new measurement of a probe to be separately recorded; in either case, the average value measured by the probe will be computed. The syntax of this pseudo-statement is:

```
$$ OFF AVERAGE
or $$ OFF NONAVERAGE
```

The AVERAGE option determines that only the average and standard deviation are kept, whereas NONAVERAGE specifies that each new value of the probe is also to be recorded and output.

The OFF pseudo-statement ends the scope of the last defined new probe, therefore avoiding possible ambiguities due to the overlapping of probes. Nesting of probes is allowed, however.

The RESULT Pseudo-statement

This pseudo-statement generates instructions to output the gathered data onto a data file. It should be used after the last executable statement of the main program, although this is not required. The syntax of this pseudo-statement is:

```
$$ RESULT
```

How to Use the Pseudo-statements

The following steps describe the use of the technique:

1. identify the number of sections of code to be monitored and select an equal number of probe names;
2. in the variable declaration part of the main program, insert the VAR pseudo-declaration listing all of the selected probe names;
3. for each section of code to be monitored, insert the ON pseudo-statement at its beginning, and the OFF pseudo-statement at its end, specifying the AVERAGE/NONAVERAGE option;
4. insert the INIT pseudo-statement before the first executable statement of the main program;
5. insert the RESULT pseudo-statement after the last executable statement of the main program;
6. execute the analyzer using the segmented version of the program as the input file and assign a new output file to hold the instrumented version;
7. compile, link and execute the instrumented version;
8. the results of measuring the

execution will be stored in a standard file.

Special Notes

The user should be aware of the general rule that all paths which begin at an ON pseudo-statement must pass through the corresponding OFF pseudo-statement. This requirement is one of the difficulties associated with analyzing programs which contain jumps, and hence the "structured" languages provide more assistance in checking that this rule is followed. The following examples illustrate the use of the ON and OFF pseudo-statements with three different PASCAL language constructs.

Example 1:

```
REPEAT
...
$$ ON PROBEL;
...
UNTIL ... ;
...
$$ OFF AVERAGE
```

is incorrect, whereas

```
REPEAT
...
$$ ON PROBEL;
...
$$ OFF AVERAGE
UNTIL ... ;
```

is correct.

Example 2:

```
IF ... THEN
BEGIN $$ ON PROBEL; S1
END
ELSE
BEGIN S2; $$ OFF AVERAGE
END ;
```

is incorrect, whereas

```
IF ... THEN
BEGIN $$ ON PROBEL; S1;
$$ OFF AVERAGE
END
ELSE
BEGIN $$ ON PROBE2; S2;
$$ OFF AVERAGE
END ;
```

is correct.

Example 3:

```
$$ ON PROBEL ;
...
IF ... THEN GOTO 1 ;
...
$$ OFF AVERAGE ;
```

is incorrect, whereas

```
$$ ON PROBEL ;
...
IF ... THEN GOTO 1 ;
...
1: ...
```

\$\$ OFF AVERAGE ;
is correct.

THE EXPERIMENT TO DETERMINE THE EXECUTION COSTS

In this section, we will discuss how experiments may be conducted to determine approximate relative costs - in units of time - for the standard operations, functions and procedures of the programming language in use.

Wortman [7] has conducted a number of experiments to compare "system time" and "hardware time"; system time was defined as the timing information returned by a logical clock (26.04 microsecond resolution) maintained for a task by IBM's OS/360 MVT operating system; hardware time was the reading obtained from the hardware clock (1 microsecond resolution). The "system time" described is very similar to the virtual CPU time maintained for each process by the VAX/VMS operating system.

Wortman concluded that the system time had a "normalized standard deviation that was, on the average, two orders of magnitude larger than the normalized standard deviation observed for hardware time" [7]. He also found that the mean value for both measurements differed by less than 0.5%.

This method of measurement was adapted for the DEC VAX-11/780 computer, using a system service routine (\$GETJPI) which gives access to the virtual CPU time [8]. For our implementation, this routine was used to estimate the cost of the various operations in standard PASCAL. Figure 4 is a schematic version of the algorithm used.

THE ANALYZER

The analyzer developed uses the self-metric approach described above, inserting tallying code which produces estimated execution times for the total run and for each segment of code that the user has chosen to monitor.

The PASCAL [16] language was chosen because of its growing acceptance in many programming situations, its elegance and, most of all, its extensive use of structured statements.

The analyzer searches the PASCAL code for the occurrence of reserved words, all standard (as well as a small number of non-standard) identifiers and operators, and determines where to insert code to account for the execution of every statement. By making use of a "cost" table, tallying statements are generated to increment "cost" counters, and the accumulated values are recorded at the end of the execution run. The Appendix describes the code which is

inserted for each PASCAL statement type.

To estimate the total execution time, which is defined as the sum of the execution times of each individual statement, it is first necessary to obtain estimates of the relative cost (in units of time) for the individual statement types. The algorithm used to do this, described in the previous section, lacks some accuracy, since it does not take into account the code optimization capability of the compiler. However, the relative costs derived, employing the most general sample statements possible, have shown themselves to be consistent, reliable, and satisfactory for their intended use as detectors of changes in performance.

Trying to incorporate the effect of the compiler optimization would introduce a variable which is too volatile, or perhaps totally uncontrollable; therefore, even at the cost of some inaccuracy, a decision was made to keep all of the study in a high level language environment.

Restrictions and Extensions

An effort was made to make the analyzer accept the entire PASCAL language as defined in [16], but the following problem was encountered:

The feature which causes some difficulty (not part of the original language definition, but in common use) is the use of externally declared procedures or functions. This is used quite frequently in large software systems, to aid modularity and to reduce the time needed for modification and compilation. This feature is also necessary to enable the use of installation defined routines existing in system libraries. Since such routines are not available to the analyzer, and need not be written in PASCAL, it is not possible for us to apply self-metric analysis to them. Therefore, the analyzer has been implemented to recognize such procedures and functions, but to take no further action.

As a result of this decision, it is not possible for us to allow procedures or functions as parameters, since such procedures or functions may be declared either within the program (and so should be analyzed), or as external routines (and so should not be analyzed). While this restriction has not limited our use of the analyzer, it may be relaxed to forbid instead the use of any externally declared routine as a parameter.

COMPARISON DATA FOR TWO ROUTINES

The most desirable characteristic of

estimated by our analyzer and the costs given by the system clock show the same order of proportion, with a percentage relative difference of about 40 per cent.

Sequential/Binary

Case	Estimated Time	Clock Time
One	0.4/1.5= 0.27	6/29= 0.20
Two	34.4/1.5= 22.9	1237/29= 42.6
Three	17.3/1.6= 10.8	631/37= 17.0

A COMPLETE EXAMPLE

A Sample Execution

A relatively small PASCAL program consisting of 8 modules was chosen to illustrate the use of the execution time estimator. This program appeared in Welsh and Elder [17].

The segmented version of the program (that is, the original source code plus the pseudo-statements inserted by the user) is shown in Figure 5. Each procedure, plus the main program, is being monitored. On entry to each procedure a probe is turned "ON"; and on exit from each procedure, the same probe is turned "OFF". The eighth probe monitors the main module of the program.

The probe associated with procedure "COPY" was turned "OFF", with the option "NONAVERAGE" to specify that the execution time estimate for that procedure is to be output each time the procedure is executed. All other probes are terminated with the "AVERAGE" option, and so will record only a summary of results.

```

three: repeat 100 times
  assign a key equal to a random
  number between the smallest and
  the largest element of the array;
  find a match for the search key
  using the binary and sequential
  search routines;

```

It is easy to see that the binary search procedure is independent of the value of the input, that is, independent of the search key. This observation is confirmed by both the estimator and the system clock, as shown below. Also as expected, the sequential search procedure is shown to be highly dependent on the particular value of the search key.

We compare the figures obtained for the sequential search procedure (on the left of each pair) with those for the binary search procedure. Although the figures obtained from the clock are subject to the variations previously noted, and the figures obtained from the analyzer are based on approximations of the time required to execute PASCAL operations, the costs

estimated by our analyzer and the costs given by the system clock show the same order of proportion, with a percentage relative difference of about 40 per cent.

Sequential/Binary		
Case	Estimated Time	Clock Time
One	0.4/1.5= 0.27	6/29= 0.20
Two	34.4/1.5= 22.9	1237/29= 42.6
Three	17.3/1.6= 10.8	631/37= 17.0

A COMPLETE EXAMPLE

A Sample Execution

A relatively small PASCAL program consisting of 8 modules was chosen to illustrate the use of the execution time estimator. This program appeared in Welsh and Elder [17].

The segmented version of the program (that is, the original source code plus the pseudo-statements inserted by the user) is shown in Figure 5. Each procedure, plus the main program, is being monitored. On entry to each procedure a probe is turned "ON"; and on exit from each procedure, the same probe is turned "OFF". The eighth probe monitors the main module of the program.

The probe associated with procedure "COPY" was turned "OFF", with the option "NONAVERAGE" to specify that the execution time estimate for that procedure is to be output each time the procedure is executed. All other probes are terminated with the "AVERAGE" option, and so will record only a summary of results.

Interpretation of the Output Data

The output of the measures generated by the tallying code of the instrumented version of the analyzed program is shown in Figure 6.

The "intermediate results" show every final value of each probe for which the NONAVERAGE option was specified in the \$\$ OFF pseudo-statement; each result displays the name of the probe and its measured value.

The "final results" give a summary of the activities of each probe. The "frequency count" gives the number of times that a segment of code was monitored by the corresponding probe. This shows how heavily the code was executed, and hence indicates its importance in relation to other monitored sections and the total program. A frequency count of zero shows that that particular code was not executed with the input data set used, and thus implies that an additional test case should be used to execute that section of code.

"Mean" gives the average estimated

execution time for the segment of code, and "standard deviation" shows the variation in the time estimates of different executions of the same piece of code. "Maximum" shows the largest execution time measured for that segment of code. Its importance lies in the fact that, even though the average execution time of a module may have changed only slightly due to a performance ripple effect, if its maximum execution time increases disproportionately, then that module is a serious candidate for reexamination.

"Total Estimated Execution Time" gives the approximate total cost for the execution of the whole program, in units of time.

These execution time figures may be compared with those obtained from previous executions of the program to identify performance changes. They may also be compared against the performance requirements of the program to detect any violations, and to provide an early warning of possible future violations.

CONCLUSION

In this paper we have shown that measuring execution time of programs to obtain a performance index is not an easy task and requires some assumptions and approximations to be made. Our self-metric approach was developed to monitor the performance of programs. The self-metric approach implies that the execution time estimates obtained are a measure of only the time for executing the various arithmetic, logic and I/O operations encountered in the program.

The procedure for using the pseudo-statements requires the user to manually segment the program by inserting pseudo-statements that are interpreted by the analyzer. While this requirement imposes some extra work on the part of the user, it also provides the freedom to monitor any section of the program that the user considers necessary. By appropriately placing the ON-OFF switch pairs, the user is able to better observe the behavior of the program.

Two modifications can be made to the current system to deal with the following situations:

1. As it is now, the user has to keep two copies of the program: one is the original source code, and the other is the segmented version. This is a waste of storage space, which may be critical if extremely large programs are to be monitored. Construction of an analyzer to accept pseudo-comments instead of pseudo-statements will permit the same copy to be accepted by both the compiler and the analyzer.

2. A totally automatic version of

the analyzer could be made available for the cases in which the smallest unit to be measured would be a procedure or function. Procedure entry and exit points are easy to detect automatically, and such information is easily related to the use of ON-OFF switches.

The estimator was, in this work, developed for the detection of performance ripple effects, but its use can be extended to other areas such as detecting heavily used code, identifying code which has not been executed by any test case, defining the relative importance of different modules, and helping in selecting between different algorithms.

APPENDIX

Instrumentation of PASCAL Statements

This appendix describes standard PASCAL statements and their instrumented versions as constructed by means of the analyzer. First we make the following definitions:

- S : Any PASCAL language statement ;
- exp : Any expression that, when evaluated, yields a value of some scalar type (e.g. integer, boolean);
- c : statement cost counter ;
- cost(x) : cost function for "x", in units of time;
- ovrhd : additional cost, not associated with any syntax unit (these are constants, determined by experiment);
- var : Any control variable;
- rec var : Any record variable;
- [x]* : indicates that "x" may occur zero or more times;
- [x | y] : indicates that either "x" or "y" will appear.

The begin - end bracket

standard syntax:

begin [S-i ;]* S-n end ;

A "begin-end" pair encloses a sequence of statements that are all executed, in order of appearance. The cost of each statement should be added to the counter before execution of that statement.

instrumented version:

```
begin [c := c + cost(S-i) ; S-i ;]*
      c := c + cost(S-n) ; S-n ;
end ;
```

cost of this statement: none.

The repeat statement

standard syntax:

```
repeat [S-i ;]* S-n
until exp ;
```

The statement sequence S-1,...,S-n is repeatedly executed until "exp", evaluated after each execution of the statement sequence, becomes true. The cost of this sequence is computed in a manner analogous to that described for the "begin-end" statement, but must also include the cost for the evaluation of the expression.

instrumented version:

```
begin
  repeat
    [c := c + cost(S-i) ; S-i ;]*
    c := c + cost(S-n) ; S-n ;
    c := c + cost(exp) + ovrhd-2
  until exp ;
  c := c + ovrhd-3
end ;
```

cost of this statement:
ovrhd-1.

The case statement

standard syntax:

```
case exp of
  [case label list-i : S-i ;]*
  case label list-n : S-n
end ;
```

The case statement selects for execution the statement whose label is equal to the current value of the selecting expression. The cost of evaluating the expression is added to that of the statement that brackets the case statement; the cost of each statement S-1,...,S-n is incorporated in the statement itself.

instrumented version:

```
case exp of
  [case label list-i :
    begin c := c + cost(S-i) ;
      S-i ;
      c := c + ovrhd-2
    end ;]*
  case label list-n :
    begin c := c + cost(S-n) ;
      S-n ;
      c := c + ovrhd-2
    end
end ;
```

cost of this statement:
cost(exp) + ovrhd-1.

The if statement

standard syntax:

```
if exp then S-1 [else S-2 | ] ;
```

The expression is evaluated only once, and the action taken next depends on its outcome: S-1 is executed if "exp" is true and S-2 if "exp" is false. The cost of evaluating the expression is added to the cost of the statement within which the "if" statement appears; the cost of S-1 and S-2 are incorporated into the statements themselves.

instrumented version:

```
if exp
  then begin c := c + ovrhd-1 ;
    c := c + cost(S-1) ; S-1 ;
    c := c + ovrhd-2
  end
  [else begin c := c + ovrhd-3 ;
    c := c + cost(S-2) ; S-2 ;
    c := c + ovrhd-4
  end | ] ;
```

cost of this statement:
cost(exp).

The for statement

standard syntax:

```
for var := exp-1 [to | downto] exp-2 do
  S ;
```

The cost of evaluating the initial expression "exp-1" and the final expression "exp-2", each evaluated only once, is added to the cost of the statement within which the "for" statement appears; the cost of the statement "S" is incorporated into the statement itself.

instrumented version:

```
begin for var := exp-1
  [to | downto] exp-2 do
  begin c := c + ovrhd-2 ;
    c := c + cost(S) ; S ;
    c := c + ovrhd-3
  end ;
  c := c + ovrhd-4
end ;
```

cost of this statement:
cost(exp-1) + cost(exp-2) + ovrhd-1.

The with statement

standard syntax:

```
with rec var [,rec var]* do S ;
```

All costs associated with evaluating the record variable(s) are added to the statement within which the "with"

statement occurred; the cost of the S statement is incorporated into itself.

instrumented version:
 with rec var [,rec var]* do
 begin c := c + cost(S) ; S end ;

cost of this statement: none.

The while statement

standard syntax:
 while exp do S ;

The controlling expression is evaluated before each iteration, therefore the cost of its evaluation is added to the cost of the statement; the total cost is incorporated into the statement itself.

instrumented version:
 begin
 while exp do
 begin c := c + ovrhd-2 ;
 c := c + cost(S) ; S ;
 c := c + cost(exp) + ovrhd-3
 end ;
 c := c + ovrhd-4
 end ;

cost of this statement:
 cost(exp) + ovrhd-1.

Procedure declaration

standard syntax:
 procedure proc-ident
 ([formal par list |] ;
 proc-body ;

Procedures may be called from different points in the program where different probes (counters) may be active, therefore, in order to increment the correct counter so that the cost of executing the procedure is added to the cost of the segment in which the call originated, it is necessary to pass the relevant probe as a variable parameter to the procedure.

instrumented version:
 procedure proc-ident
 ([formal par list; |]
 var c : integer) ;
 proc-body ;

cost of this declaration: none.

Procedure statement

standard syntax:
 proc-ident [(actual par list) |] ;

The active probe (counter) has to be passed to the procedure as a parameter to account for the cost of the execution of

the statements in the procedure.

instrumented version:
 proc-ident ([actual par list, |]
 active-probe) ;

cost of this statement:
 cost(actual par list) +
 cost(procedure call).

Function declaration

standard syntax:
 function func-ident
 ([formal par list] |]
 : func-type ;
 func-body ;

The considerations for a function definition are analogous to those presented for the procedure declaration.

instrumented version:
 function func-ident
 ([formal par list; |]
 var c : integer) : func-type ;
 func-body ;

cost of this declaration: none.

Function call

standard syntax:
 func-ident [(actual par list) |] ;

The considerations for a function call are analogous to those presented for the procedure statement.

instrumented version:
 func-ident ([actual par list , |]
 active-probe) ;

cost of this statement:
 cost(actual par list) +
 cost(function call).

Goto statement

standard syntax:
 [S-i ;]* goto label ;

The cost associated with the goto statement must represent the costs of all statements executed along the path to the goto statement.

instrumented version:
 [c := c + cost(S-i) ; S-i ;]*
 c := c + cost(goto) ;
 goto label ;

cost of this statement: none.

Labelled statement

standard syntax:
 [S-i ;]* label : [S-j ;]*

The occurrence of a label causes a new count to be initiated, anticipating a later jump.

instrumented version:

```
[c := c + cost(S-i) ; S-i ;]*
label : [c := c + cost(S-j) ; S-j ;]*
```

cost of this statement: none.

REFERENCES

- [1] Ferrari, D., Computer Systems Performance Evaluation, Prentice-Hall, Englewood Cliffs, N.J., 1978.
- [2] Ingalls, D.H., "FETE - A FORTRAN Execution Time Estimator", in Program Style, Design, Efficiency, Debugging, and Testing (ed. D. Van Tessel), Appex. II, Prentice-Hall, Englewood Cliffs, N.J., 1974.
- [3] Yau, S.S., Collofello, J. and MacGregor, T., "Ripple Effect Analysis of Software Maintenance", Proc. Compsac 78 - Comp. Software and Applications Conf., Nov. 1978, pp. 60-65.
- [4] Yau, S.S. and Collofello, J., "Performance Ripple Effect Analysis for Large-Scale Software Maintenance", Interim Technical Report, RADC-TR-80-55, March 1980, NTIS AD-A084-351.
- [5] Crowley, C., "The Architecture of Clocks", ACM SIGARCH, Vol. 7, No. 11, Dec. 1979, pp. 4-9.
- [6] Davies, J., "Clock Architecture and Management", ACM SIGARCH, Vol. 8, No. 5, Aug. 1980, pp. 3-6.
- [7] Wortman, D.B., "A Study of High-Resolution Timing", IEEE Trans. on Software Engineering, Vol. SE-2, June 1976, pp. 135-137.
- [8] DEC Corporation, VAX/VMS Systems Services Reference Manual, Vol. 4, DEC Corp., Maynard, MA, March 1980.
- [9] Lyon, G. and Stillman, R., "Simple Transforms for Instrumenting FORTRAN Programs", Software - Practice and Experience, Vol. 5, 1975, pp. 347-358.
- [10] Ferguson, L., "Profile : An Automated Program Analysis Aid", ACM Sigmetrics - Conf. on Comp. Performance : Modeling, Measurement and Management, 1977.
- [11] Ramamoorthy, C.V., Kim, K.H. and Chen W.T., "Optimal Placement of Software Monitors Aiding Systematic Testing", IEEE Trans. on Software Engineering, Vol. SE-1, No. 4, Dec. 1975, pp. 403-410.
- [12] Knuth, D. and Stevenson, F., "Optimal Measurement Points for Program Frequency Counts", BIT, Vol. 13, 1973, pp. 313-322.
- [13] Cheung, R.C. and Ramamoorthy, C.V. : "Optimal Measurements of Program Path Frequency and its Applications", Proc. of the Sixth Intern. Federation of Automatic Control Congress, Boston, 1975, pp. 1-6.
- [14] Stucki, L., "Automatic Generation of Self-Metric Software", Proc. 1973 IEEE Symp. on Computer Software Reliability, 1973.
- [15] Yau, S.S., Ramey, J.L. and Nicholl, R.A., "Assertion Techniques for Dynamic Monitoring of Linear List Data Structures", Jour. Systems and Software, Vol. 1, No. 4, 1980, pp. 319-336.
- [16] Jensen, K. and Wirth, N., PASCAL User Manual and Report, Springer-Verlag, N.Y., 2nd. ed., 1979.
- [17] Welsh, J. and Elder, J., Introduction to PASCAL, Prentice-Hall International, Englewood Cliffs, N.J., 1979.

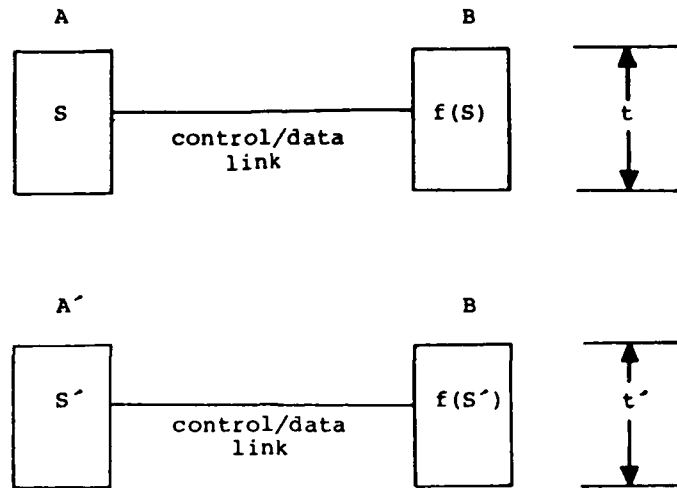


Fig. 1 - Schematic representation of performance ripple effect

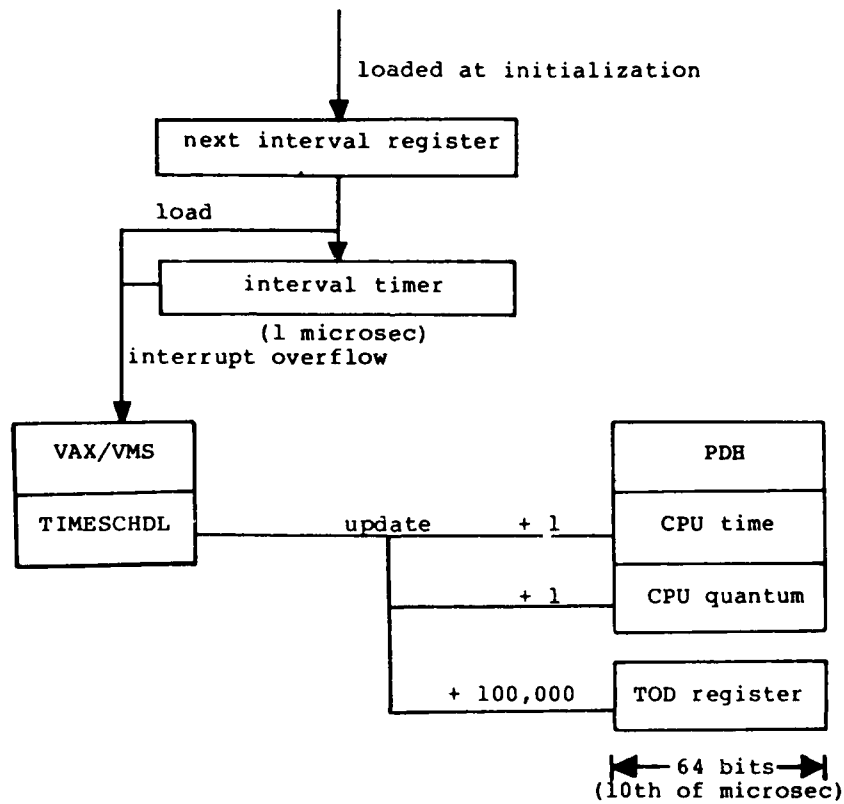


Fig. 2 - Schematic representation of the VAX timing system

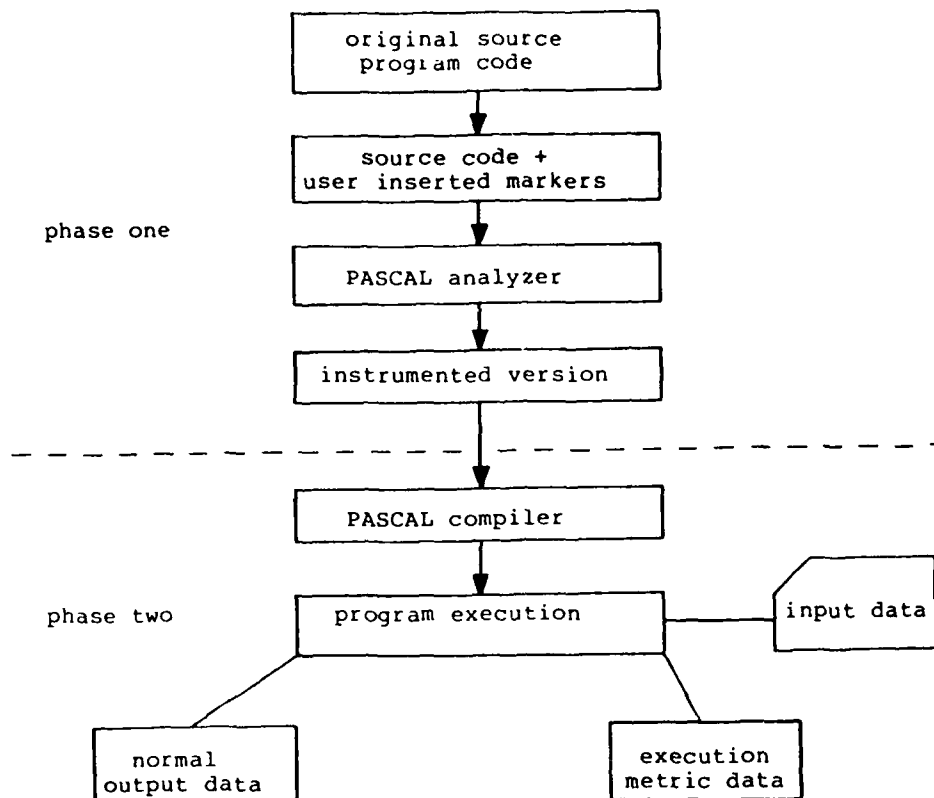


Fig. 3 - Simplified view of the self metric approach

The algorithm:

```

VAR I,J,K : INTEGER ;
    CLOCK1, CLOCK2, DIFF : INTEGER ;
    MEAN, DEV : REAL ;

PROCEDURE $GETJPI (VAR TIME: INTEGER); EXTERN ;

BEGIN
  FOR I := 1 TO NUMBEROFMEASUREMENTSESSIONS DO
    BEGIN RESET VARIABLES ;
      FOR J := 1 TO NUMBEROFSAMPLES DO
        BEGIN $GETJPI (CLOCK1) ;
          FOR K := 1 TO NUMBEROFITERATIONS DO
            EXECUTE SAMPLE STATEMENT ;
            $GETJPI (CLOCK2) ;
            DIFF := CLOCK2 - CLOCK1 ;
            COMPUTE DYNAMIC MEAN AND
              STANDARD DEVIATION FOR 'DIFF' ;
          END ;
        OUTPUT RESULTS ;
      END ;
    END ;
  END ;

```

Fig. 4 - Algorithm used to estimate the cost of operations

```

PROGRAM SORT (DATAFILE,OUTPUT);

CONST MAXKEY = 99999;

TYPE KEYTYPE = 0..MAXKEY;
  SOMETYPE = RECORD TRANSTYPE : (DELIVERY,DISPATCH);
    AMOUNT : 1 .. MAXINT
  END;
  ITEM = RECORD KEY : KEYTYPE;
    RESTOFRECORD : SOMETYPE
  END;
  FILETYPE = FILE OF ITEM;

VAR C, DATAFILE : FILETYPE;

(* DEFINE 8 PROBES *)
$$ VAR CXZ1,CXZ2,CXZ3,CXZ4,CXZ5,CXZ6,CXZ7,CXZ8;

PROCEDURE NATURALMERGESORT(VAR C : FILETYPE);
VAR NUMBEROFRUNS : 0 .. MAXINT;
  A,B : FILETYPE;
  ENDOFRUN : BOOLEAN;

  PROCEDURE COPY(VAR SOURCE, DESTINATION : FILETYPE);
  VAR COPIEDITEM : ITEM;
  BEGIN $$ ON CXZ1;
    COPIEDITEM:=SOURCE^;
    GET (SOURCE);
    DESTINATION^:=COPIEDITEM;
    PUT (DESTINATION);
    IF EOF(SOURCE) THEN
      ENDOFRUN := TRUE
    ELSE ENDOFRUN := COPIEDITEM.KEY > SOURCE^.KEY;
    $$ OFF NONAVERAGE
  END; (*COPY*)

  PROCEDURE COPYARUN(VAR SOURCE,DESTINATION: FILETYPE);
  BEGIN $$ ON CXZ2;
    REPEAT
      COPY(SOURCE,DESTINATION);
    UNTIL ENDOFRUN;
    $$ OFF AVERAGE
  END; (*COPYARUN*)

  PROCEDURE DISTRIBUTE;
  BEGIN $$ ON CXZ3;
    REPEAT
      COPYARUN(C,A);
      IF NOT EOF(C) THEN COPYARUN(C,B);
    UNTIL EOF(C);
    $$ OFF AVERAGE
  END; (*DISTRIBUTE*)

  PROCEDURE MERGE;

  PROCEDURE MERGEARUNFROMAANDB;
  BEGIN $$ ON CXZ4;
    REPEAT
      IF A^.KEY<B^.KEY THEN
        BEGIN COPY(A,C);
          IF ENDOFRUN THEN COPYARUN(B,C);
        END
      ELSE BEGIN COPY(B,C);
          IF ENDOFRUN THEN COPYARUN(A,C);
        END;
    END;
  END;

```

Fig. 5 - A sample segmented program


```

        UNTIL ENDOFRUN;
        $$ OFF AVERAGE
    END; (*MERGEARUNFROMAANDB*)

    BEGIN $$ ON CXZ5;
        WHILE NOT (EOF(A) OR EOF(B)) DO
            BEGIN MERGEARUNFROMAANDB;
                NUMBEROFRUNS:=NUMBEROFRUNS + 1;
            END;
            WHILE NOT EOF(A) DO
                BEGIN COPYARUN(A,C);
                    NUMBEROFRUNS:=NUMBEROFRUNS + 1;
                END;
            WHILE NOT EOF(B) DO
                BEGIN COPYARUN(B,C);
                    NUMBEROFRUNS:=NUMBEROFRUNS + 1;
                END;
            $$ OFF AVERAGE
        END; (*MERGE*)

    BEGIN $$ ON CXZ6;
        REPEAT RESET(C);
            REWRITE(A); REWRITE(B);
            DISTRIBUTE;
            RESET(A); RESET(B);
            REWRITE(C);
            NUMBEROFRUNS:=0;
            MERGE;
        UNTIL NUMBEROFRUNS=1;
        $$ OFF AVERAGE
    END; (*NATURALMERGESORT*)

    PROCEDURE COPYFILE(VAR F,G:FILETYPE);
    BEGIN $$ ON CXZ7
        RESET(F);
        REWRITE(G);
        WHILE NOT EOF(F) DO
            BEGIN WRITELN(F^.KEY);
                G^:=F^;
                PUT(G); GET(F);
            END;
        $$ OFF AVERAGE
    END; (*COPYFILE*)

    BEGIN
        (* INITIALIZE THE COUNTERS *)
        $$ INIT;

        $$ ON CXZ8;
        WRITELN('^**UNSORTED RECORD KEYS **^');
        COPYFILE(DATAFILE,C);
        NATURALMERGESORT(C);
        WRITELN; WRITELN;
        WRITELN('^** SORTED RECORD KEYS **^');
        COPYFILE(C,DATAFILE);
        WRITELN('^ END OF PROGRAM^');
        $$ OFF AVERAGE;

        (* PRINT FINAL EXECUTION RESULTS *)
        $$ RESULT
    END.

```

Fig. 5 - A sample segmented program (continued)

*** INTERMEDIATE RESULTS ***

```
CXZ2      =      4490
CXZ2      =      8970
CXZ2      =     13438
CXZ2      =      4478
CXZ2      =     13438
CXZ2      =     13450
CXZ2      =     13438
CXZ2      =      4478
```

*** FINAL RESULTS ***

PROBE	FREQUENCY	COUNT	MEAN	STD. DEVIATION	MAXIMUM
1	24		4274.400	4.752	4277
2	8		9522.501	3221.567	13450
3	2		27806.500	119.250	28045
4	2		20442.000	3373.500	27189
5	2		28008.500	39.750	28088
6	1		131220.000	0.000	131220
7	2		31301.000	0.000	31301
8	1		197138.000	0.000	197138

TOTAL ESTIMATED EXECUTION TIME : 197138

Fig. 6 - Sample Execution Time Output for Program SORT.

DESIGN STABILITY MEASURES FOR SOFTWARE MAINTENANCE*

Stephen S. Yau
 Department of Electrical Engineering
 and Computer Science
 Northwestern University
 Evanston, Illinois 60201

and

James D. Johnston
 Computer Science Department
 Arizona State University
 Tempe, Arizona 85287

SUMMARY

The high cost of software during its life cycle can be attributed largely to software maintenance activities, and a major portion of these activities is to deal with the modifications of the software. In this paper, design stability measures which indicate the potential ripple effect characteristics due to modifications of the program at the design level are presented. These measures can be generated at any point in the design phase of the software life cycle which enables early maintainability feedback to the software developers. The validation of these measures and future research efforts involving the development of a user-oriented maintainability measure which incorporates the design stability measures as well as other design measures are discussed.

Index Terms - Design stability measures, program modifications, and software maintenance.

INTRODUCTION

The major expenses in computer systems at present are in software. While the cost of hardware is decreasing rapidly, software productivity improves only slowly. Thus, the cost of software relative to hardware is rapidly increasing. The majority of this software cost can be attributed to software maintenance. The cost of maintenance activities has been very high ranging from 40 percent to as high as 80 percent of the total cost during the life cycle of large-scale software systems [Boeh73, Zelk78, Lien78].

The control of software maintenance costs can be approached in several ways. One approach is to improve the productivity of maintenance practitioners by providing them with tools and techniques to help them perform their maintenance tasks. Advances in this area have included debugging tools, program flow-charters, and ripple effect analysis tools. Although these

tools are definitely important in the maintenance phase, a more effective approach to reducing maintenance costs is to develop tools and techniques which are applicable during the earlier life cycle phases which give program developers into producing more maintainable software. These tools and techniques include a broad range of design methodologies and programming techniques.

Another approach to controlling software maintenance costs is the utilization of software metrics during the program development phase [Henr81, Silb77]. Many metrics have been developed which assess different quality characteristics of software. These metrics can be utilized as indicators of program quality, and can help identify potential problem areas in programs. More effort is needed in the development of measures for evaluating the maintainability of a program at each phase of its development.

In this paper, we will discuss an approach to reducing maintenance costs through the utilization of metrics. We will first discuss the software maintenance process and the software quality attributes that affect the maintenance effort. Because accommodating the ripple effect of modifications in a program is normally a large portion of the maintenance effort, especially for not well designed programs [Yau84], we will present some measures for estimating the stability of a program design. The stability of a program design is the primary attribute indicating the resistance to the potential ripple effect which a program developed from the design would have when it is modified. These design stability measures can be obtained at any point in the design process and, thus, enable examination of programs early in their life cycle for possible maintenance problems. Algorithms for computing these design stability measures will be presented in detail. Applications of these measures, an illustrative example, some experimental results for an indirect validation will also be presented. Future research efforts involving the development of a user-oriented maintainability measure will also be discussed.

* This work was supported by Rome Air Development Center, U.S. Air Force System Command under Contract No. F30602-80-C0139.

The Maintenance Process

Software maintenance is a very broad activity that includes error corrections, enhancements of capabilities, deletion of obsolete capabilities, and optimization [Yar78]. We have previously modelled this maintenance process as shown in Figure 1 [Yar80]. The first phase of the maintenance process consists of analyzing a program in order to understand it. This phase is affected by the complexity, documentation, and self-descriptiveness of the program. The second phase consists of generating a particular modification proposal to accomplish the implementation of the maintenance objective. This phase is affected by the extensibility of the program. The third phase consists of accounting for the ripple effect as a consequence of program modifications. The primary attribute affecting ripple effect is the stability of the program, where stability is defined as the resistance to the amplification of changes in the program. The fourth phase consists of testing the modified program to ensure the modified program has at least the same reliability as before. This phase is affected by the testability of the program.

Each of these four phases and their associated software quality attributes are critical to the maintenance process. Several metrics have already been developed for some of these phases applicable to a program coding. These metrics include program complexity metrics [Yar79], and, to a lesser extent, extensibility and testability metrics [Yar79]. The utilization of these types of metrics during the program design phase can help identify potential maintenance problem areas.

Although the above-mentioned metrics can be designed to provide important maintainability information, the program restructuring needs to improve maintainability can be quite high during the coding phase of the life cycle. A more cost-effective approach would be the application of these types of metrics during the program design phase. Any restructuring necessary to improve maintainability could then be accomplished in order than restructuring later. Several metrics applicable during the design phase which contribute to program maintainability have been developed. For example, metrics exist for evaluating the complexity [Beard, Whit80] and testability [Yar80] of software designs. Mobile connectivity metrics such as coupling, strength, fan-in, fan-out, and control-effectiveness-control which are also applicable for software designs have been developed [Beard, Mpe79, Yar79].

Although the above design metrics contribute towards the evaluation of program maintainability, they do not directly address the quality attribute of design stability. Design stability measurement requires a more in-depth

analysis of the interrelationships that are provided by current programming practices. The nature of design stability measurement and how it relates to program design will now be described.

Design Stability Measurement

In this section, the design stability measured will be presented in detail. The measures are then applied to the well-known design notions of data abstraction and data hiding introduced by Barua [Bar76]. These notions can have a profound impact on the maintainability of a program. The lack of adequate documentation and inadequate assumptions, leading program modifications, if changes are made which affect these notions, a ripple effect may occur through the program requiring additional costly changes [Yar81].

The design stability measure will be derived from an examination of those assumptions existing in the program. The design stability of a module will be measured as the magnitude of the potential ripple effect as a result of modifying the module. The potential ripple effect will be defined as the total number of assumptions made by that module, which either invoke the module, use a module, are defined, or shared with other modules, within the module, or are otherwise used by the module. It implies that modules with few assumptions are likely to affect only a few other modules, and hence have a small ripple effect. The design stability of a program will be defined as the magnitude of the total ripple effect of all modules in the program. It implies that the calculation of the design stability of any part of the program can be done.

It should be noted that the design stability of a module is not a function of the structure of the program. It is a function of a module's use, and hence, the design stability of a module is a function of the assumptions made by the module. The design stability of a program is a function of the assumptions made by all modules in the program, as well as the environment in which the program will execute. Even when these assumptions are identified, they are normally only specified in comments in the documentation. This problem has long been recognized and partially eliminated by modern programming practices which

allow the examination of assumptions implemented in the code. The design stability measure of the program is a function with assumptions which were stated explicitly in the code. The design stability of a program is a function of the assumptions made by a particular module of the assumptions made by all modules in the program, as well as the environment in which the program will execute. Even when these assumptions are identified, they are normally only specified in comments in the documentation. This problem has long been recognized and partially eliminated by modern programming practices which

and into the black box concept of modules. The black box concept approaches that the assumptions primarily concerned with module interfaces must be more explicit [Yorl9].

The calculation of the design stability measure will be restricted to those assumptions concerning the interfaces. This restriction is motivated in light of modern programming techniques and the extremely difficult task of identifying and specifying all assumptions. The calculation of assumptions concerning module interfaces is more accurate than current design stability, such as that of module interface analysis [Yorl9] which only examines module interfaces, or module coupling measures [Lynch, Yorl9] which examine module interfaces without analyzing the assumptions associated with these interfaces. The design stability measure described in this paper exceeds the ability of these design measures by incorporating the assumptions associated with these interfaces.

The identification of the assumptions associated with a module's interface requires initially the identification of a module's interface. A module's interface is defined to consist of the module's passed parameters, global variables, and shared files. This incorporates the concept of module coupling [Yorl9] and naturally extends it by including shared files. Shared files are included as part of a module's interface because of their potential structural intermodule communication. In order to calculate the module interface more precisely, each parameter, global variable, and shared file will be examined to see if it is composed of other identifiable entities. For example, a record can be decomposed into its respective fields and other structural data types into their respective basic types. Decomposition criterion simply tries to define these minimal entities for which a program can be specified. Thus, if a record data type is part of a module's interface and that record consists of a character, an integer and a real number, then the three minimal entities are the character, the integer, and the real number. Separate assumptions concerning each of these entities could then be set included.

In order to set interface and simplify the recording of the assumptions made by each module about its minimal interface entities, two categories of assumptions will be utilized. The first type of assumption concerns the basic type of the entity such as integer, real, boolean, character, etc. This assumption is always recorded and can be checked automatically by a compiler or other interface compilation type checking tool [Tich9]. The second type of assumption concerns the value of the entity and is recorded if the module has any assumptions about the value which the minimal entity may assume. Since some records cannot make any assumptions about the value of their

interface variables, these records type of assumptions may not always be defined.

For example, a module which is passed an integer may not have any restrictions upon its permissible values for the integer, and thus, may not possess any assumptions about its entity category. In a strongly typed language such as Pascal, the type declaration of a module entity may include restrictions on the entity's values analogous to those of category type assumptions. For example, a character type includes assumptions about both the type and value of the entity. In order to facilitate across programming language type declaration which include restrictions on values will be recorded as both category and value type assumptions.

Since the design stability measure is calculated by counting assumptions about interfaces, each minimal entity in an interface will contribute a maximum of one assumption to each category. The parameters, global variables, and shared files in an interface which are composed of these minimal entities will all contribute to this assumption count. For example, an array of integers defined as part of an interface implies an assumption about its structure. This assumption should be counted towards that of the module's interface. The minimal entity for this structure is an integer which implies a maximum of two one assumptions may be made concerning this interface. Thus, a total of three assumptions may be recorded for the array of integers in the module's interface. In general, each structured data type in an interface should be decomposed into its basic type and one assumption for the structure recorded. The basic type of the interface should then be examined and additional assumptions recorded. As another example, consider an array of students where a student is a record consisting of an ID number and a grade. Assumptions may be recorded for the array structure, the record structure, and the ID number and grade for a maximum of six assumptions. This method of recording assumptions is far more detailed than simply counting parameters as evidenced in the preceding example.

AN ALGORITHM FOR COMPUTING DESIGN STABILITY MEASURES

In this section an algorithm will be presented for calculating the design stability measures for a program module. This algorithm upon the assumptions concerning the module's interfaces developed in the previous section. Basically, the design stability of a module will be computed as the reciprocal of the total number of assumptions made by other modules concerning the module whose stability is being measured. The algorithm consists of the following steps:

Step 1: From the program design documentation, analyze the module invocation hierarchy for the program and for each module x , identify the following set:

- (a) J_x (invoker when invoke module x),
- (b) J'_x (invoker invoked by module x),
- (c) J_x (parameters returned from module x to module y , where $y \in J_x$),
- (d) J'_x (parameters passed from module x to module y , where $y \in J'_x$).

Step 2: From the program design documentation, analyze the program's global data which is defined to consist of global variables and shared files, and for each module x identify the following set:

- (a) G_x (global data referenced in module x),
- (b) G'_x (global data defined in module x).

From these sets, for each global data item i , identify the set

$$G_x = \{y \mid i \in G_x, y \in J_x\}$$

Step 3: For each set R_{xy} and each parameter $i \in R_{xy}$, find the number of assumptions made by module y about i utilizing the following pseudo-code algorithm:

If parameter i is a structured data object, then decompose i into its base types and increment the assumption count by 1, else consider i to be a minimal entity.

While i is base elements can be decomposed, consider i as an element which is not a minimal entity, and decompose it into its base elements and increment the assumption count by 1.

For a minimal entity comprising i , if module y makes assumptions about the values which the minimal entity may assume, then increment the assumption count by 2, else increment the assumption count by 1.

Let TP_{xy} equal to the total number of assumptions made by y about the parameters in R_{xy} .

Step 4: For each set K'_x and each parameter $i \in K'_x$, find the number of assumptions made by module y about i utilizing the pseudo-code algorithm in Step 3.

Let TP'_{xy} equal to the total number of assumptions made by y about the parameters in K'_x .

Step 5: For each module x and every global data item $i \in G_x$, find the number of assumptions made about i by other modules in the program. This requires utilization of the set G_x and application of the algorithm in Step 3 for each global data item i and every module $y \in G_x - \{x\}$.

Set TG_x equal to the total number of assumptions made by other modules about the global data items in G_x .

Step 6: For each module x , compute the design logical ripple effect $DLRE_x$ as follows:

$$DLRE_x = TG_x + \sum_{y \in J_x} TP_{xy} + \sum_{y \in J'_x} TP'_{xy}$$

Step 7: For each module x , calculate the design stability DS_x as follows:

$$DS_x = 1/(1 + DLRE_x)$$

Step 8: Compute the program design stability PDS as follows:

$$PDS = 1/(1 + \sum_x DLRE_x)$$

where x is a module in the program.

AN EXAMPLE

In this section, computation of the design stability measures will be illustrated for two designs for the same problem. This example is taken from 'Your79'. The problem consists of reading text from an online keyboard and text in a card file, dissecting the text into words, and combining these words according to code from the keyboard and codes contained in the cards. Input begins with the keyboard and continues, character-by-character, until the ideograph "SRC" is received. At that point, the reading of input from cards is to commence and continue until the ideograph "ll" is reached. Input from the keyboard then resumes. An end-of-transmission from the keyboard triggers reading the remaining cards. The continuous stream of text from these two sources is to be broken into separate English words, which are then passed individually to a pre-existing module named PBOOKWORD. The high level structure charts for each module with corresponding inputs and outputs are shown in Figure 2 for alternative 1 and Figure 3 for alternative 2.

The design stability algorithm will now be applied for both alternatives.

AD-A143 763

METHODOLOGY FOR SOFTWARE MAINTENANCE(U) NORTHWESTERN
UNIV EVANSTON IL S S YAU FEB 84 RADDC-TR-83-262
F30602-80-C-0139

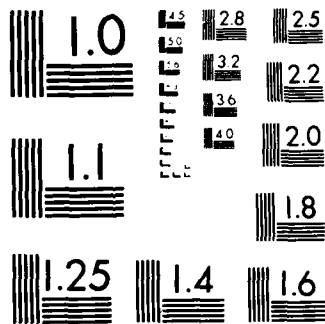
4/4

UNCLASSIFIED

F/G 9/2

NL





MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS 1963-A

ALTERNATIVE 1Step 1

$J_{SCANWORD} = 0,$
 $J'_{SCANWORD} = \{1, INKEY, READCARD, FINDWORD, PROCWORD\}$
 $J_{INKEY} = J_{READCARD} = J_{FINDWORD} = J_{PROCWORD} = \{SCANWORD\},$
 $R_{INKEY, SCANWORD} = \{\text{character, end-of-transmission flag}\},$
 $R_{READCARD, SCANWORD} = \{\text{card image, last-card flag}\},$
 $R_{FINDWORD, SCANWORD} = \{\text{word, end-of-words flag, get-character flag, get-card flag, word-done flag}\},$
 $R_{PROCWORD, SCANWORD} = \emptyset,$
 $R'_{SCANWORD, INKEY} = R'_{SCANWORD, READCARD} = \emptyset,$
 $R'_{SCANWORD, PROCWORD} = \{\text{word}\},$
 $R'_{SCANWORD, FINDWORD} = \{\text{character, end-of-transmission flag, card image, last-card flag, source}\}.$

Step 2 There are no global data items.

Step 3

$TP_{INKEY, SCANWORD} = 1 + 1 = 2,$
 $TP_{READCARD, SCANWORD} = 2 + 1 = 3,$
 $TP_{FINDWORD, SCANWORD} = 2 + 1 + 1 + 1 + 1 = 6,$
 $TP_{PROCWORD, SCANWORD} = 0.$

Step 4

$TP'_{SCANWORD, INKEY} = TP'_{SCANWORD, READCARD} = 0,$
 $TP'_{SCANWORD, PROCWORD} = 2,$
 $TP'_{SCANWORD, FINDWORD} = 1 + 1 + 2 + 1 + 1 = 6.$

Step 5 $TG_x = 0$ for all modules x .

Step 6

$DLRE_{SCANWORD} = 8,$
 $DLRE_{INKEY} = 2,$
 $DLRE_{READCARD} = 3,$
 $DLRE_{FINDWORD} = 6,$
 $DLRE_{PROCWORD} = 0.$

Step 7

$DS_{SCANWORD} = 1/9,$
 $DS_{INKEY} = 1/3,$
 $DS_{READCARD} = 1/4,$
 $DS_{FINDWORD} = 1/7,$
 $DS_{PROCWORD} = 1.$

Step 8 $PDS_{Alternative 1} = 1/20.$

ALTERNATIVE 2Step 1

$J_{SCANTEXT} = \emptyset,$
 $J'_{SCANTEXT} = \{\text{GETWORD, PROCWORD}\},$
 $J_{GETWORD} = J_{PROCWORD} = \{\text{SCANTEXT}\},$
 $J'_{GETWORD} = \{\text{GETCHAR, GETCARD}\},$
 $J_{GETCHAR} = J_{GETCARD} = \{\text{GETWORD}\},$
 $J'_{PROCWORD} = J'_{GETCHAR} = J'_{GETCARD} = \emptyset,$
 $R_{GETWORD, SCANTEXT} = \{\text{word, end-of-words flag}\},$
 $R_{PROCWORD, SCANTEXT} = \emptyset,$
 $R_{GETCHAR, GETWORD} = \{\text{character, end-of-transmission flag}\},$
 $R_{GETCARD, GETWORD} = \{\text{word, end-of-words flag}\},$
 $R'_{SCANTEXT, GETWORD} = \emptyset,$
 $R'_{SCANTEXT, PROCWORD} = \{\text{word}\},$
 $R'_{GETWORD, GETCHAR} = R'_{GETWORD, GETCARD} = \emptyset.$

Step 2 There are no global data items.

Step 3

$TP_{GETWORD, SCANTEXT} = 2 + 1 = 3,$
 $TP_{PROCWORD, SCANTEXT} = 0,$
 $TP_{GETCHAR, GETWORD} = 1 + 1 = 2,$
 $TP_{GETCARD, GETWORD} = 2 + 1 = 3.$

Step 4

$TP'_{SCANTEXT, GETWORD} = 0,$
 $TP'_{SCANTEXT, PROCWORD} = 2,$
 $TP'_{GETWORD, GETCHAR} = TP'_{GETWORD, GETCARD} = 0.$

Step 5 $TG_x = 0$ for all modules x .

Step 6

$DLRE_{SCANTEXT} = 2,$
 $DLRE_{GETWORD} = 3,$
 $DLRE_{PROCWORD} = 0,$
 $DLRE_{GETCHAR} = 2,$
 $DLRE_{GETCARD} = 3.$

Step 7

$DS_{SCANTEXT} = 1/3,$
 $DS_{GETWORD} = 1/4,$
 $DS_{PROCWORD} = 1,$
 $DS_{GETCHAR} = 1/3,$
 $DS_{GETCARD} = 1/4.$

Step 8

$PDS_{Alternative 2} = 1/11.$

Analysis of the metrics obtained for both alternatives indicates that alternative 2 is more stable than alternative 1. This finding is supported by the discussion in the source of the example that alternative 2 is easier to program and maintain. Further analysis of these metrics indicates that the primary sources of instability in alternative 1 are modules FINDWORD, and SCANWORD. This finding is again supported by the discussion in the source of the example [Your 79].

VALIDATION

An important requirement of any metric is the capability of validating it. In this section both direct and indirect approaches to validating the design stability measures will be discussed. A direct approach to validation consisting of experimentation with the metrics was performed by the authors utilizing a graduate software engineering class. The class consisted of 24 professional programmers with diverse company experiences. The course assignment was to design and implement an automated gradebook system in PASCAL. The class was divided into 4 teams each of which was to build a program of an estimated 4K lines. The class utilized the structured design methodology to produce a complete program design specification. This design specification was then utilized to compute the design stability measures. The module design stability measures obtained had a broad range from 1/145 to 1. It was interesting to note that the degree of module fan-in/fan-out did not always correlate with the design stability. For example, many modules with small fan-in/fan-out had poor stability and vice versa.

Upon completion of the program design specification, the class was then asked to submit proposals for possible changes to the program. Over 200 such change proposals were received. These proposals were analyzed in terms of their potential ripple effect if they were to be implemented. Several interesting results of this experiment will now be described.

The first result is that those modules which would have contributed large ripple effects if modified are among the modules possessing poor design stability measures. The converse, however, is not necessarily true. Since the design stability measures reflect a potential worst case ripple effect, it is possible for modules with poor stability to be modified in certain ways without producing a large ripple effect.

Another result of the experiment illustrated the diagnostic capabilities of the design stability measures. Many of the modules found to possess poor stability also were of weak functional strength and were common coupled to many other modules. It should be noted, however, that some modules which possess poor stability are not necessarily bad. For example, implementations of data abstractions usually possess

poor stability. The important point is that if the assumptions made upon a module with poor stability are violated, the potential ripple effect is large. Thus, these assumptions must be examined carefully with an eye towards future modifications.

Although the experimentation with the design stability measures produced several interesting results, it cannot be utilized as a complete validation of the measures. Experiments with maintenance-type measures can be very misleading due to the diverse and numerous types of maintenance tasks which may be performed. For example, maintenance data collected regarding the maintenance activity that a particular program experienced may not be representative of the maintenance activity in other programs. A complete direct validation of the design stability measures will, thus, require a large database of maintenance information for a significant number of various types of programs which have undergone a sufficient number of modifications of a wide variety. The short-term possibility of utilizing such a maintenance database for validating maintenance-type measures is not very promising. In light of this reality and diverse nature of the maintenance tasks performed by users of software systems, a more user-oriented approach to maintenance metric computation is needed. These user-oriented maintainability metrics will combine the unique potential future maintenance requirements of a user with the characteristics of the software associated with these potential modifications to produce a tailored measure of the expected maintainability to be experienced by the user. These ideas will be described in more detail later.

Since further experimentation utilizing the design stability measures could be misleading without a large maintenance database, a complete direct validation will be delayed until the development of a user-oriented maintainability measure. The design stability measures can be, however, indirectly validated by arguing how the measures are affected by various already established attributes of programs which affect maintainability. It should be noted that most of these established attributes suffer from the same validation problems as the design stability measures, and their acceptance is largely a consequence of intuitive arguments.

Because one program attribute which affects maintainability is the utilization of data abstraction and information hiding [Parn77], an indirect validation of the design stability measures must show that the design stability of programs utilizing data abstraction and information hiding is generally better than that of programs which do not. Since our measures are based upon counts of assumptions made concerning interface variables and since a lack of data abstraction and information hiding manifests itself in an increase in assumption counts, it is apparent that the design stability

of programs utilizing data abstraction and information hiding is generally better than that of programs which do not.

The relationship of the design stability measures with both the data abstraction and global variable notions can be further illustrated by the following example:

Consider the case of 3 modules A, B, and C which share a global array of records, where each record consists of an integer ID number and a real balance as indicated in Figure 4. If we also assume that no parameters are passed between the MAIN module and modules A, B, and C and that modules A, B, and C make assumptions about the values of the ID number and the balance, the following values can be obtained:

$$DS_{MAIN} = 1,$$

$$TG_A = TG_B = TG_C = 6 + 6 = 12,$$

$$DLRE_A = DLRE_B = DLRE_C = 12,$$

$$DS_A = DS_B = DS_C = 1/13,$$

$$PDS = 1/37.$$

In Figure 5, the program is redesigned to utilize a data abstraction module X to eliminate the need for having a global array of records. The data abstraction passes a single record to the modules A, B, and C depending upon some index variable. From the design, the following values may be obtained:

$$DS_{MAIN} = 1,$$

$$TG_A = TG_B = TG_C = 0,$$

$$TP'_{AX} = TP'_{BX} = TP'_{CX} = 3 + 2 = 5$$

(assuming that X makes no assumptions about the values in the record)

$$TP_{XA} = TP_{XB} = TP_{XC} = 5,$$

$$DLRE_A = DLRE_B = DLRE_C = 5,$$

$$DLRE_X = 15,$$

$$DS_A = DS_B = DS_C = 1/6,$$

$$DS_X = 1/16,$$

$$PDS = 1/31.$$

These two examples illustrate the detrimental effect of global data on stability as well as the positive effect of data abstraction modules. The data abstraction modules, although quite unstable themselves, improve the stability of the modules which utilize them.

The design stability measures presented here can, thus, be indirectly validated since they incorporate and reflect some aspects of

program design generally recognized as contributing to the development of program stability during maintenance.

APPLICATIONS OF THE DESIGN STABILITY MEASURES

The design stability measures presented in this paper can be utilized for comparing alternative designs of a module or program at any point in the design phase of the software life cycle. The selection of alternatives which exhibit favorable design stability measures can lead to more maintainable programs.

The design stability measures can also be utilized to identify portions of the program which exhibit poor stability and, thus, may contribute to ripple effect problems during the maintenance phase. These portions of the program can be easily identified by the measures and examined for deficiencies. Those areas of the program with poor stability can then be redesigned incorporating such favorable design approaches as abstraction, information hiding, restriction of global variables and functionality in order to improve the design stability measures.

The design stability measures will also be a key component of any overall maintainability measure. As previously discussed, stability is an important attribute of program maintainability which must be combined with other attributes in order to formulate a maintainability measure. Thus, our future research efforts in the development of a user-oriented maintainability measure will incorporate these design stability measures.

CONCLUSIONS AND FUTURE RESEARCH

In this paper, measures for estimating design stability of a program and of the modules within a program have been presented. Algorithms for computing these design stability measures, applications of these measures, an illustrative example, some experimental results, and an indirect validation of the measures have also been presented.

Much research remains to be done in this area. Our primary emphasis will be on the development of a user-oriented maintainability measure computable during the design phase of the software life cycle. This metric will incorporate our design stability measure as well as design complexity and testability measures. Much experimentation will be needed in combining these quality attributes into a single measure. Extensive validation on large-scale programs will also be performed.

ACKNOWLEDGEMENT

The authors would like to express their appreciation for the helpful discussions with A. Bowles and S. C. Chang.

REFERENCES

- [Gilb77] T. Gilb, Software Metrics, Winthrop Publishers, Inc., 1977.
- [Hals77] M. H. Halstead, Elements of Software Science, New York: Elsevier North-Holland, 1977, pp. 84-91.
- [Hendr81] S. Henry and D. Kafura, "Software Structure Metrics Based on Information Flow", IEEE Trans. on Software Engineering, Vol SE-7, No. 5, Sept. 1981, pp. 510-518.
- [Meyer75] G. J. Myers, Reliable Software Through Composite Design, Petrocelli/Charter, 1975.
- [McCa76] T. J. McCabe, "A Complexity Measure", IEEE Trans. Software Eng., Vol. SE-2, Dec. 1976, pp. 308-320
- [Parn72] D. L. Parnas, "On the Criteria to be Used in Decomposing Systems into Modules", Communications of the ACM, Vol. 15, No. 12, December 1978, pp. 1053-1058.
- [Tich79] W. F. Tichy, "Software Development Control Based on Module Interconnection", in Proc. Fifth International Conference on Software Engineering, pp. 29-41.
- [Whit80] M. H. Whitworth and P. A. Szulewski, "The Measurement of Control and Data Flow Complexity in Software Designs", Proc. COMPSAC 30, Oct. 1980, pp. 735-743.
- [Yau78] S. S. Yau, J. S. Collofello, and T. M. Mac Gregor, "Ripple Effect Analysis of Software Maintenance", Proc. COMPSAC 78, Nov. 1978, pp. 60-65.
- [Yau80] S. S. Yau and J. S. Collofello, "Some Stability Measures for Software Maintenance", IEEE Trans. on Software Engineering, Vol. SE-6, No. 6, Nov. 1980, pp. 545-552.
- [Yin80] B. H. Yin, "Software Design Testability Analysis", Proc. COMPSAC 80, Oct. 1980, pp. 729-734.
- [Your79] E. Yourdon and L. L. Constantine, Structured Design, Prentice-Hall, 1979.
- [Zelk78] M. V. Zelkowitz, "Perspectives on Software Engineering", ACM Computing Surveys, Vol. 10, June 1978, pp. 197-216.

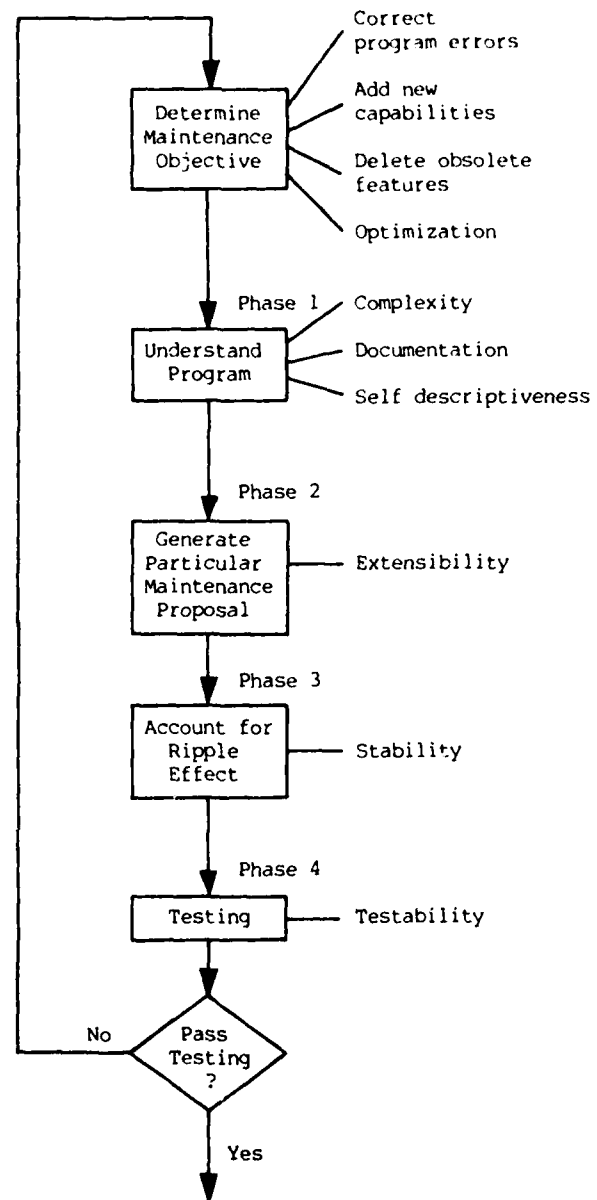
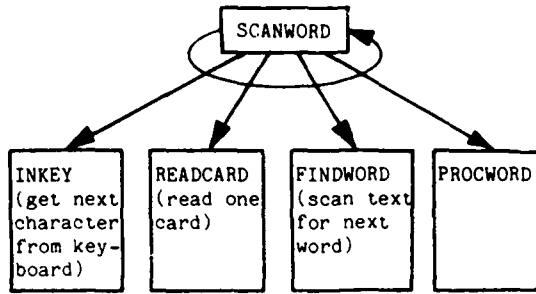


Figure 1. The software maintenance process and associated program quality attributes in each phase.



MODULE	INPUTS	OUTPUTS
INKEY		character, end-of-transmission flag
READCARD		card image, last-card flag
FINDWORD	character, end-of-transmission flag, card image, last-card flag, source	word, end-of-words flag, get-character flag, get-card flag, word-done flag
PROCWORD	word	

Figure 2. Design for Alternative 1.

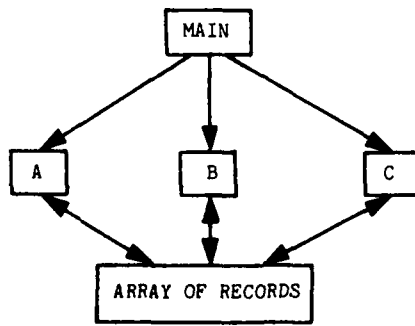
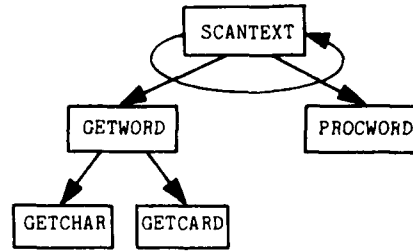


Figure 4. An illustration of modules A, B, and C sharing a global array of records.



MODULE	INPUTS	OUTPUTS
GETCHAR		character, end-of-transmission flag
GETCARD		card-image, last-card flag
GETWORD		word, end-of-words flag
PROCWORD	word	

Figure 3. Design for Alternative 2.

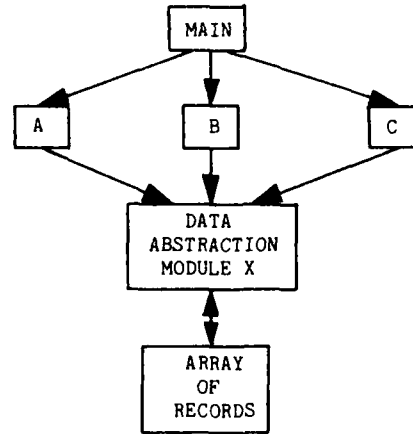


Figure 5. An illustration of modules A, B, and C utilizing a data abstraction module to access an array of records.

Reprinted from THE PROCEEDINGS OF INTERNATIONAL COMPUTER SYMPOSIUM 1982
Taiwan, December 15 - 17, 1982.

A METHODOLOGY FOR SOFTWARE MAINTENANCE*

Stephen S. Yau, Carl K. Chang**, Chung-Chu Hsieh**,
Zenichi Kishimoto+ and Robin A. Nicholl

Department of Electrical Engineering and Computer Science
Northwestern University
Evanston, Illinois 60201, U.S.A.

ABSTRACT

The rapidly increasing cost of software maintenance indicates the importance of developing effective methodologies for software maintenance. In this paper a methodology for software maintenance, which decomposes the software maintenance process into four phases is presented. The first phase is to analyze the software in order to understand it. The second phase is to generate and realize a particular modification proposal. The third phase is to account for all of the ripple effects of the modification, including both logical and performance ripple effects. The fourth phase is to test the modified program to insure that it functions properly. To support a wide spectrum of activities involved in these four phases, a variety of software tools have been developed. By making use of these tools, an environment has thus been created to assist the software maintenance practitioners in performing their functions more effectively. Currently, these tools have not been totally integrated. A method for integrating these tools and the databases they need into a unique maintenance environment is presented. Most of the tools discussed in this paper have been demonstrated for PASCAL on a DEC VAX-11/780 computer.

Index Terms- software maintenance, program representation, program modification, program editor, program slicing, ripple effect analysis, test

*This work was supported by the Rome Air Development Center, U.S. Air Force System Command, under Contract F30602-80-C-0139.

**Carl K. Chang and Chung-Chu Hsieh are now with Bell Telephone Laboratories, Naperville, Illinois 60540, U.S.A.

+Zenichi Kishimoto is now with GTE Laboratories, Waltham, Massachusetts 02154, U.S.A.

case generation

INTRODUCTION

Most of the expenses associated with computer systems are due to the cost of developing and maintaining software. The total U.S. expenditure on programming in 1977 was estimated at between \$50 and \$100 billion, which represents more than 3% of the U.S. GNP for that year [1]. It has been estimated that by 1985, the cost of computer software will soar to 90% of the total system expenditure [2]. This is due to the dramatically decreasing cost of hardware and the increasing complexity and cost of software, which has required ever greater human resources to develop, validate and maintain.

It is well recognized that the maintenance cost of software has increased continuously and that it has become the single dominant cost item during the life cycle of a large-scale software system. Estimates of maintenance cost have been found ranging from 40% [2], 67% [3], to as high as 80% [4] of the total cost during the life cycle of large-scale software systems. Therefore, in order to reduce the high cost of software, it is essential to develop effective software maintenance methodologies.

"Software maintenance" has been defined as "the process of modifying existing operational software while leaving its primary functions intact" [5]. The broad spectrum of activities which comprise software maintenance includes error corrections, enhancements of capabilities, deletion of obsolete capabilities, optimization, and minor changes in mission requirements [6]. An excellent review of the state-of-the-art software maintenance techniques and tools can be found in [7]. As indicated in that report, much more attention has been focused on the management aspects of

maintaining software systems than on the technical aspects. Maintenance programmers must still handle the technical problems in an ad hoc manner.

Therefore there is an urgent need for an effective software maintenance methodology, which should not only address all the major problems of software maintenance, but also provide a well-integrated maintenance environment to effectively solve the software maintenance problems. In this paper, we will discuss a methodology for software maintenance which incorporates a variety of software tools to support a unified maintenance environment. Most of the tools mentioned in this paper have been demonstrated for PASCAL on a DEC VAX-11/780 computer. Each of these tools, however, operates on its own representation of programs. A method for integrating these tools into a software maintenance environment will also be discussed.

OVERVIEW OF THE METHODOLOGY

Yau et al. [8] have presented an integrated view of the software maintenance process. Once a particular maintenance objective has been established, the objective can be accomplished in the four phases as shown in Figure 1.

The first phase is to analyze the program in order to understand it. To facilitate this, the requirements, the different levels of the design and the program itself should be clearly described. This description of the software system can be best prepared during software development when each level of the software system and its connection with other levels are understood. Since there are many software systems currently in operation which have been developed without such descriptive support, it is necessary for us to establish a procedure by which the information can be constructed by analysis of the existing programs, using only the source code, in addition to whatever documentation is available.

The second phase is to generate a particular maintenance proposal so that the maintenance objective can be achieved. The multi-level system description mentioned in phase 1 can be used to determine the effects of the maintenance objective on each of the levels. A given program modification for an existing program must be specified at different levels (i.e. requirement, specification, design and code levels).

The code-level specification of the modification can then be realized [9] in the program code to produce a modified program which is subject to re-validation.

The third phase is to account for all the ripple effect of the modifications proposed in phase 2. As a result of these modifications, there may be logical inconsistencies and/or significant degradation in program performance. The ripple effect analysis technique [10-16], will identify both logical and performance ripple effects of the proposed program modification.

The fourth phase is to test the modified program to insure that it functions correctly. During the software maintenance phase, it is important that cost-effective testing techniques are applied [17]. Testing of the modified software must be done in order to detect unexpected errors, such as dormant errors which, although present in the software system before the modification, may become active errors as a result of the modification.

If the modified program fails to pass the testing phase, any or all of the previous phases must be repeated, depending on the extent and type of failure. In the most extreme case, the maintenance objective may itself be considered infeasible (because of its maintenance cost, for example), and should be altered.

SOFTWARE MAINTENANCE PROCESS

In this section, we are going to discuss each of the four phases of the software maintenance process in more detail.

Understanding the Software

During the maintenance phase of the A-7 aircraft flight program [18], Heninger found that the existing documentation was sparse and not up-to-date. Therefore, she decided that it would be more cost-effective to re-construct the software requirements before attempting to modify the software. This software maintenance example indicates the importance which maintenance personnel place on a good description of the software system. It also shows that, even after the software system has entered the operational phase, it is still feasible to construct such a description.

In general, the maintenance personnel should not be required to understand the entire system at a detailed level, because of the cost and time required to do so. We prefer an approach which allows changes to a software system to be made correctly, with the effort to understand the system being concentrated on only the portions of the software system relevant to the modification. To meet this goal, a detailed description of the software system should be available, which would also record the relationships of various components at different levels (requirements, architectural design, detailed design and program code). When such a layered description of the software system is available, tracing changes to particular portions of the software system may be done more easily and more accurately.

A further benefit of such a description is the ability to directly relate the program code of the software system to the modification request, which is often expressed in terms which are more familiar to the users of the system than to the maintenance programmers. This relationship is usually unclear if only the program itself is available. However, some useful descriptive information can still be extracted from the source code of the programs alone, using automated analysis tools.

Program analysis tools have been available for many years to provide aids such as automatic flowcharting and construction of call graphs. Since graph representations of data or execution flow are often used to describe software system requirements (as in RSL [19], for example), we must analyze the program so that we can present our information in such a format. Under these circumstances, the most attractive approach to the construction of this software system description is one based on data flow analysis of the program, with the intention of relating inputs and outputs of the program to each other. The "program slicing" technique [20] can be used for this purpose.

Program slicing refers to a process of selecting a portion of the text of a program to form a "slice", where the selection is done automatically, based on data flow analysis techniques. The user of a program slicer must specify which variables are of importance, and at which point in the program their values are of interest. These two pieces of information constitute the "slicing criterion". The program slicer uses the slicing criterion to analyze the data

flow of the program to extract any code which may contribute to the values of those variables at that point. These program slices are themselves syntactically correct programs and, if executed, will produce values equal to those produced by the original program at the selected point (assuming that the original program contains no non-terminating loops [20]).

Generating and Realizing Modification Proposals

When a number of "change requests" from the users are collected for attention by the maintenance staff, a "modification" is started. There are a number of ways to implement a particular modification, and each of these is known as a "modification proposal" until we have selected one particular modification proposal to achieve the maintenance objective. The elements which make up a modification are "program changes". To generate a modification proposal, it is necessary to carry out activities similar to those of requirements, design and coding, as performed during the development phase. The change requests are assumed to be in an informal notation, but the maintenance staff must (ultimately) alter a software system which is precisely expressed or written in a formal language. Therefore, there is a need to convert the modification from an informal notation to a formal one.

We have chosen to attack this problem from both directions: from the direction of the informally defined changes and from the direction of the formally defined software. First, we need a method whereby we can relate each item informally mentioned in each change request to some known entity in the software system. In addition, we must determine how the new behavior required of those items may be formally described to generate a formal modification proposal to the software system. We call this process the specification of a modification proposal. Figure 2 shows the relationship between the level of the modification proposal and the level of the software description.

The following steps are repeated for each level of the modification proposal:

1. Identify the description level to which the modification applies.
2. Define the interface between each change and the software system.

3. Trace the effects of each change at this level.
4. Restructure the software system to reduce extraneous effects.
5. Tentatively make the modification.
6. Check the correctness of the modification.
7. Refine the modification proposal by decomposing each change into one or more changes to the next level.

The effects of the changes can be traced at a particular level of description by performing "ripple effect" analysis on a model of that level of description, and at lower levels, using a definition of the interface between each level of description. SREM [19] provides some tracing information describing the preparation of the software system requirements, and some information about the inter-connections between different requirements, but this is not adequate for our detailed analysis, nor is it relevant to the software design or code.

In addition, since most software systems are written in well-defined programming languages, we can define a formal model of a software system and a set of formal operations on that model. This means that a modification proposal can be stated as a set of formal modification operations, which are to be applied to the software system to implement that modification proposal. We call this process the realization of a modification proposal.

Program modification of an existing software system must be carried out by physically modifying the software at the code level, either by correcting the existing program code or by developing new segments of program code [21]. Modifying programs, however, is an incremental process. We have developed techniques to assist programmers in modifying only the relevant portions of the program and in re-asserting its correctness with a minimal amount of re-analysis of the program [22]. Incremental program modification should be conducted interactively, so that maintenance programmers can expect instant feedback on the effects of the modification, and thus be able to make program modifications more intelligently. This approach is obviously advantageous, because the length of the fix-and-compile cycle is shortened. After the program has been "fixed", it is ready for ripple

effect analysis (phase 3) and testing (phase 4).

Ripple Effect Analysis

An important factor contributing to the high cost and complexity of software maintenance is that the effects of program modification are usually not restricted to the location of the modification, but propagate to other portions of the program. This phenomenon has been fully described in [10] and is called the "ripple effect" of program modification. Ripple effect analysis techniques have been developed for analyzing two aspects of these ripple effects, the logical or functional aspect and the performance aspect. Logical ripple effect analysis involves the identification of program areas which may require additional maintenance to ensure the logical or functional consistency of the software. Performance ripple effect analysis involves the identification of performance repercussions throughout the software system as a result of the changes to one program area.

We have made an extensive study of logical ripple effect analysis techniques [10-13]. The phenomenon of logical ripple effects is a serious problem for maintenance programmers who must modify large-scale software systems since the repercussions from their modifications are rarely obvious. Our automated technique to perform logical ripple effect analysis is based on a model of the data and control dependencies which exist in programs. We extend the data flow to include not only USED and DEFINED sets, but also a mapping to show how variables are used to define other variables. This model is called an "error flow" model, since it shows the means by which potential errors may propagate through a program.

When a modification is made to a program, changes occur in the data flow of the program. A set of variables, known as the primary error source set, is directly affected by the modification. Our ripple effect tracing algorithms use the arcs of the data flow graph to determine where the effects of the primary error source set may reach, and hence all the potential logical ripple effects of the modification are identified.

Logical ripple effect analysis can be decomposed into two stages. The first stage is the information construction stage, where both the intramodule error flow model and the intermodule error flow model are constructed. The second stage

is the error flow tracing stage. Two difficulties associated with logical ripple effect analysis are further caused by recursion and dynamic aliasing, due to the fact that logical ripple effect analysis is based on a static analysis of the data flow properties of the program. Both problems have recently been solved under certain reasonable assumptions [13].

We have also initiated the study of performance ripple effect analysis techniques [14-16]. Since large-scale software systems often have strict performance requirements, it is also important to insure that program modifications do not degrade program performance.

When modifications are made to a program, performance ripple effects occur as well as logical ripple effects. We have developed a model of the ways in which performance ripple effects may propagate as a result of program modification. In this model we identify attributes of the program which affect its overall performance. These attributes are quantifiable measures of performance. The most obvious example of a performance attribute is the execution time of module.

By identifying the performance dependency relationships which exist between performance attributes, we can construct a complete model of potential performance ripple effect propagation. When a change is made to a section of the program code, certain performance attributes may be affected. These performance attributes may, in turn, affect other performance attributes in the program due to a performance dependency relationship. Performance dependency relationships are created as a result of certain mechanisms in the program. For example, calling a module is a mechanism which creates a performance dependency relationship from the called module to the calling module. Specifically, this means that the execution time attribute of the called module may affect the execution time of the calling module.

In [15] we describe a number of performance attributes and the possible performance dependency relationships between them. Using this model we have also developed algorithms to trace the potential performance ripple effects from an initial modification. These algorithms are also presented in [15]. Yau et al [23] refined some of these techniques and verified the basic formulation of the approach using an

automated tool.

Effective Testing for Software Maintenance

After all the modifications and their ripple effects have been accommodated, testing is performed. Testing is done to validate the modified program in order to detect unexpected errors due to the modifications, such as previously dormant errors which may have become active errors due to the modification. A complete testing strategy for the maintenance phase consists of module testing, integration testing, and system function testing. We have concentrated on a module testing technique which is part of an overall testing strategy for software maintenance. This technique uses the input partition method for test case generation and the data-driven symbolic evaluation method for test case execution [17].

For each of the modified modules, test cases are generated by comparing the detailed specifications and the program code. Whenever possible, we will use test cases in the original test set which go through any modified portion of the program. However, it is also necessary to generate additional test cases. These test cases are then used to evaluate the behavior of the modified software. Our approach is to use symbolic execution, driven by actual test case data, to produce symbolic test results. In addition to test case generation and test case execution, the technique also supports debugging of the module when the existence of errors has been detected.

SOFTWARE MAINTENANCE ENVIRONMENT

In the following sections we will describe software tools which we have developed for software maintenance. These tools have been demonstrated on a DEC VAX-11/780 computer under the VMS operating system.

The Syntax-directed Program Editor

One major contribution made by syntax-directed editors is that they treat a program as a well-formed collection of syntactic units (language constructs), not just text.

We have developed a syntax-directed editor which uses three classes of editing command: basic modification commands, cursor movement commands, and extended modification commands. The basic modification commands include ADD,

INSERTA, INSERTB, DELETE and CHANGE. These commands are "basic" because they provide the basic mechanisms to enable maintenance programmers to modify programs. The cursor movement commands include UP, DOWN, LEFT, RIGHT and DIAGONAL. Making use of these cursor movement commands facilitates "structural movement" rather than "textual movement" through the program. With these commands, programmers can make more sensible moves to locate the desired constructs. The extended modification commands include CUT, PASTE, PASTEB, COPY and REPLACE. These extended commands provide further editing power for the user.

Details about the mechanism working behind these commands can be seen in [9]. This editor operates on a syntax-oriented program representation which is also fully described in [9].

An incremental analysis mechanism must be associated with the editor to evaluate the static semantics of programs. For example, the command to delete a variable declaration may trigger the invocation of a semantic checking routine which highlights all the usages of that variable, to remind the programmer of the existence of a potential semantic inconsistency.

The syntax-directed editor is also supported by a screen-oriented pretty-printer which allows the programmer to view the portion of the program being edited. The programmer first uses cursor movement commands to examine the program, then uses modification commands to modify the program. The pretty-printer responds to cursor movements commands and recognizes program changes by examining the program representation. It then rebuilds the screen display according to the change. As a result, the pretty-printer provides instant visual feedback to assist the programmer to perceive program changes in an interactive manner. Figure 3 illustrates the structural cursor movement commands.

The Syntax-directed Program Slicer

Weiser's program slicer [20] operated on a conventional form of data flow graph [24] (i.e. a directed graph whose nodes represent the conditions and assignment statements of the program and whose edges represent possible data flow paths between them).

In an interactive programming environment, and in normal practice, it is more useful to display the text

(including data declarations) of the code in the slice. We have developed a program slicer which meets these requirements. Our program slicer interactively constructs the text of a partial program (or "slice") which satisfies the slicing criterion. Each slice is a syntactically correct program, made up of a subset of the declarations and statements of the original program. To achieve this, we have extended the program representation mentioned in the previous section to include the data flow information which the program slicer needs.

Our current approach is based on an intramodule program slicer. It selects a portion of a module (i.e. procedure or function) according to the slicing criterion, and adds to it the declarations of objects inside or outside the module to insure that it forms a syntactically correct program. In PASCAL these objects include labels, constants, types, variables, procedures and functions. To enhance the usefulness of this program slicer as a programming aid, we have added options of further applying the slicer to existing slices of a program - to obtain a more refined picture of program behavior - and of combining slices (possibly those of distinct modules) into more comprehensive units. The operations which are available to combine program slices are UNION and INTERSECTION of program slices. Figure 4 illustrates how our program slicing technique works.

The Logical and Performance Ripple Effect Analyzer

A software support system, the "logical ripple effect analyzer", for performing logical ripple effect analysis on PASCAL programs has been developed. This support system consists of three subsystems: an intramodule error flow analyzer, an intermodule error flow analyzer, and a logical ripple effect identification subsystem. The intramodule error flow analyzer was developed by modifying an existing PASCAL compiler. The other two subsystems were newly developed. These programs operate on the intramodule error flow model and the intermodule error flow model.

The program analyzer developed to construct a "performance ripple effect model" for PASCAL programs has been implemented by modifying the same PASCAL compiler. A program to trace performance ripple effects has also been written, which handles initialization of the data structures in the program, user interaction, and the tracing algorithms

themselves. The "performance ripple effect analyzer" consists of these two programs.

Testing by Symbolic Execution

Our current results in software testing are limited to module testing. We have demonstrated this technique for programs written in ANSI FORTRAN since our implementation makes use of existing tools for data flow analysis (DAVE [25]) and symbolic execution (ATTEST [26]), which only operate on FORTRAN programs.

We use the DAVE data-flow analysis system [25] as a preprocessor to produce the control graph of a program to be analyzed. From this graph we use a program graph generator, which we have developed, to construct the program graph for further analysis. The tokens of the program, which are produced by the DAVE system, are used by an intermediate code generator, which is a part of ATTEST's preprocessor [26], to construct an intermediate code representation of the program. This intermediate code will be used for symbolic execution of the program.

We have also developed a modification handler to store modification information in the program graph produced by the program graph generator. The ATTEST symbolic execution system was modified to permit data driven execution, and this modified system is used for test case selection and test case execution. The results of data driven symbolic execution are used for output validation.

A test execution tool was developed to perform test execution interactively. This tool is used for debugging, and uses four types of command: test case specification commands, move commands, show commands and breakpoint commands.

Although it has been demonstrated for FORTRAN programs, this module testing technique can be adapted to block structured programming languages by altering the front-end (preprocessor) and the user interface (modification handler).

AN INTEGRATED SOFTWARE MAINTENANCE ENVIRONMENT

Before an integrated system can be achieved, experimentation must be performed, based on independent execution of each tool currently existing in our software maintenance environment. Results of these separate experiments

have been described in [9,13,17,23]. Although we are convinced that our techniques can benefit maintenance personnel in a direct fashion, further investigation into various other aspects of software maintenance is still required. Because an environment of this kind is highly experimental in nature, we must pay equal attention to tool construction and environment experimentation in the future.

Any programming environment must be highly experimental in nature [27]. The performance of each tool in a particular environment must be studied and altered accordingly in order to achieve a highly effective integrated system. The software tools described above have been demonstrated independently in order to show that their implementation was feasible. Consequently, each tool has been developed to operate on its own program model (or representation), although together they provide a wide spectrum of program analyses.

However, our experience has shown that information constructed for each independent tool can also be shared among several tools of similar nature. For example, data flow information appears both in the program representation used by the program slicer and the program editor, and in the error flow model used by the logical ripple effect analyzer. The program representation used by the program editor implicitly contains the control flow information which is essential to the module testing tools. The performance analysis tools also require information regarding control flow and data flow, although they also require additional performance oriented information.

In order to integrate all these tools to form an effective maintenance machine, several models representing various aspects of a software system may exist simultaneously. We view these different pieces of information collectively as a portion of the multi-level software system description. However, it is still necessary to develop a mechanism, whereby the maintenance activities can be carried out harmoniously and efficiently. This mechanism may be considered to be a "modification session manager", which will support a friendly user interface and effective and accurate information handling. The modification session manager has the responsibility of controlling the users' use of the different software tools in performing modification activities. Figure 3 shows how such a system can be organized.

CONCLUSION

In this paper we have presented a comprehensive software maintenance methodology. All phases contained in this methodology and techniques involved in each phase have been briefly described. The status of prototype systems based on various techniques has been discussed. Based on the framework reported here, we expect to conduct full experiments in applying our methodology to a large-scale software system in the near future.

REFERENCES

- [1] Boehm, B. W., "Software Engineering", IEEE Trans. on Computers, Vol. C-25, No. 12, December 1976, pp. 1226-1241.
- [2] Lenman, M. M., "Programs, Life Cycles, and Laws of Software Evolution", Proc. of the IEEE, Vol. 68, No. 9, September 1980, pp. 1060-1076.
- [3] Boehm, B. W., "Software and Its Impact : A Quantative Assessment", Datamation, May 1973, pp. 48-59.
- [4] Bauer, H. A. and Birchall, R. H., "Managing Large Scale Software Development with An Automated Change Control System", Proc. 2nd. Int'l. Conf. on Computer Software and Applications (COMPSAC 78), November 1978, pp. 13-18.
- [5] Zelikowitz, M., "Perspectives on Software Engineering", ACM Computing Surveys, June 1978, Vol. 10, No. 2, pp. 197-216.
- [6] Lientz, B. P. and Swanson, E. B., "Characteristics of Application Software Maintenance", Communications ACM, Vol. 24, No. 6, June 1978, pp. 466-471.
- [7] Donahoo, J. D. and Swearingen, D., A Review of Software Maintenance Technology, RADC-TR-80-13, February 1980.
- [8] Yau, S. S. and Collofello, J. S., "Some Stability Measures for Software Maintenance", IEEE Trans. on Software Engineering, Vol. SE-6, No. 6, November 1980, pp. 545-552.
- [9] Chang, C. K., "Incremental Modification of Computer Programs", Ph.D. Dissertation, Northwestern University, 1982.
- [10] Yau, S. S., Collofello, J. S. and MacGregor, T. M., "Ripple Effect Analysis of Software Maintenance", Proc. 2nd. Int'l. Conf. on Computer Software and Applications (COMPSAC 78), November 1978, pp. 60-65.
- [11] Yau, S. S., Collofello, J. S. and Hsien, C. C., Self-Metric Software - A Handbook: Part I, Logical Ripple Effect Analysis, Final Technical Report RADC-TR-80-138, Vol II (of 3), NTIS AD-A0386-291, April 1980.
- [12] MacGregor, T. M., "Analysis of Logical Ripple Effect of Program Modification", Ph.D. Dissertation, Northwestern University, 1979.
- [13] Hsien, C.-C., "An Approach to Logical Ripple Effect Analysis for Software Maintenance", Ph.D. Dissertation, Northwestern University, 1982.
- [14] Collofello, J. S., "Effect of Program Modification on Software Performance", Ph.D. Dissertation, Northwestern University, 1979.
- [15] Yau, S. S. and Collofello, J. S., "Performance Ripple Effect Analysis for Large Scale Software Maintenance", Interim Technical Report RADC-TR-80-55, NTIS AD-A084-351, March 1980.
- [16] Yau, S. S. and Collofello, J. S., "Self-Metric Software - A Handbook: Part II, Performance Ripple Effect Analysis", Final Technical Report RADC-TR-80-138, Vol III (of 3), NTIS AD-A0386-292, April 1980.
- [17] Kishimoto, Z., "Effective Software Testing for Software Maintenance", Ph.D. Dissertation, Northwestern University, 1982.
- [18] Heninger, K. L., "Specifying Software Requirements for Complex Systems", Proc. Spec. of Reliable Software, 1979, pp. 1-14.
- [19] Alford, M. W., "A Requirements Engineering Methodology for Real-Time Processing Requirements", Vol. SE-3, No. 1, Jan. 1977, pp. 60-69.
- [20] Weiser, M., "Program Slicing", Proc. 5th Int'l Conf. on Software Engineering, 1981, pp. 439-449.
- [21] Donahoo, J. and Swearingen, D., "Software Maintenance Technology", Proc. 4th Int'l. Conf. on Computer Software and Applications (COMPSAC 78), November 1978, pp. 60-65.

- 80), 1980, pp. 394-400.
- [22] Yau, S. S., Chang, C. ... and Nicholl, R. A., "An Approach to Incremental Program Modification", submitted for publication.
- [23] Yau, S. S., Carvalho, M. B. and Nicholl, R. A., "A Method for Estimating the Execution Time of Arbitrary Paths in Computer Programs", Proc. 5th Int'l. Conf. on Computer Software and Applications (COMPSAC 81), November 1981, pp. 225-239.
- [24] Hecht, M. S., Flow Analysis of Computer Programs, North-Holland Publishing Company, Amsterdam, 1977.
- [25] Osterweil, L. J. and Foadick, L. D., "DAVE -A Validation Error Detection and Documentation System for Fortran Programs", Software Practice and Experience, Vol. 6, No. 4, 1976, pp. 473-486.
- [26] Clarke, L. A., "A System to Generate Test Data and Symbolically Execute Programs", IEEE Trans. on Software Engineering, Vol. SE-2, No. 3, Sep. 1976, pp. 215-222.
- [27] Barstow, D. R. and Shrobe, E. E., "Observations On Interactive Programming Environments", IEEE Tutorial: Software Development Environments, 1981, pp. 286-301.

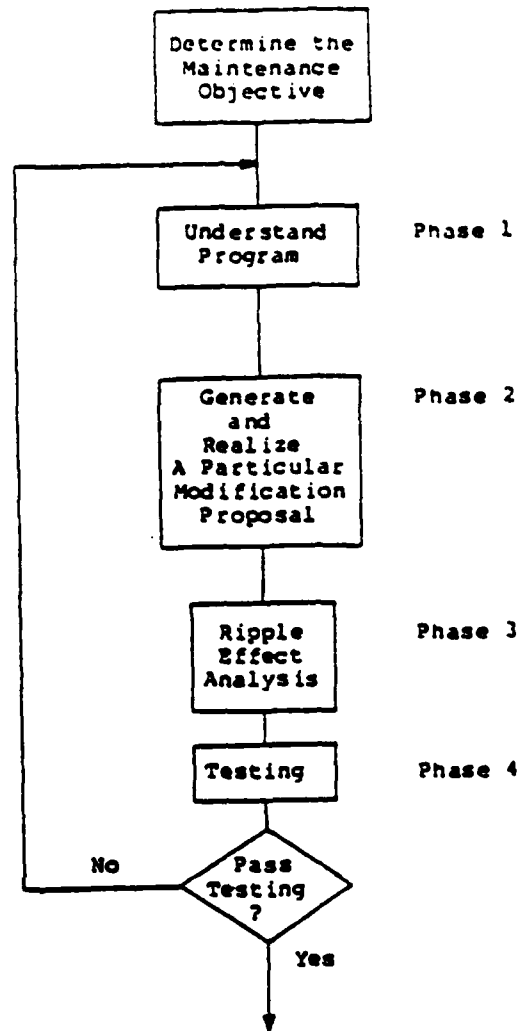


Figure 1. The Software Maintenance Process

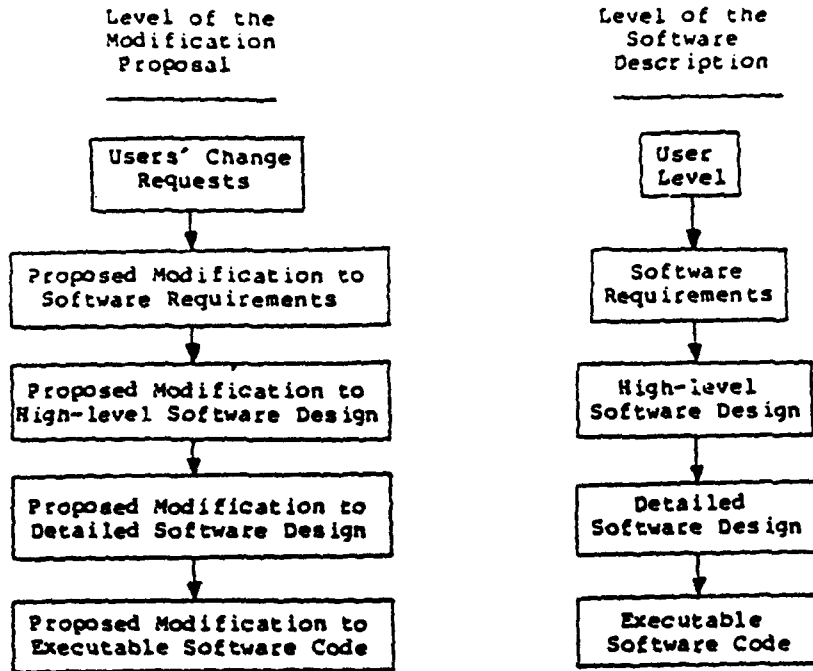


Figure 2. The approach to specifying modification proposals.

```

begin
  reset (data) ;
  while (not eof(data)) do
    begin
      while (not eoln(data)) do
        *1
        *2
        begin
          *3
          read (data, ch) ;
          *4
          case ord (ch) of
            *5
            ff : begin
                  write (ch) ;
                  read (carr)
                end ;
            cr : write (ch) ;
            lf : writeln
            otherwise write (ch)
          end
        end ;
      readln (data)
    end
  end.

```

```

position *1 to position *2 - RIGHT
         *2                 *3 - DOWN
         *3                 *4 - RIGHT
         *4                 *5 - DOWN

```

Figure 3. Structural cursor movements.

```

program triangle (input, output) ;
  { This program builds a digit triangle }

  var i, j, k : integer;

  begin
    for j := 1 to 9 do
      begin
        i := 1;
        for i := 1 to j do
          write(i:l);
          for k := j downto 2 do
            write(k-l:l);
          writeln;
        end
      end { triangle } .

```

(a)

```

program triangle (input, output) ;

  var j, k : integer;

  begin
    for j := 1 to 9 do
      begin
        for k := j downto 2 do
          end
        end
      end
    end.

```

(b)

Figure 4 (a) The program to be sliced.
 (b) An illustration of the syntax-directed
 program slicing technique (slicing for
 variable k).

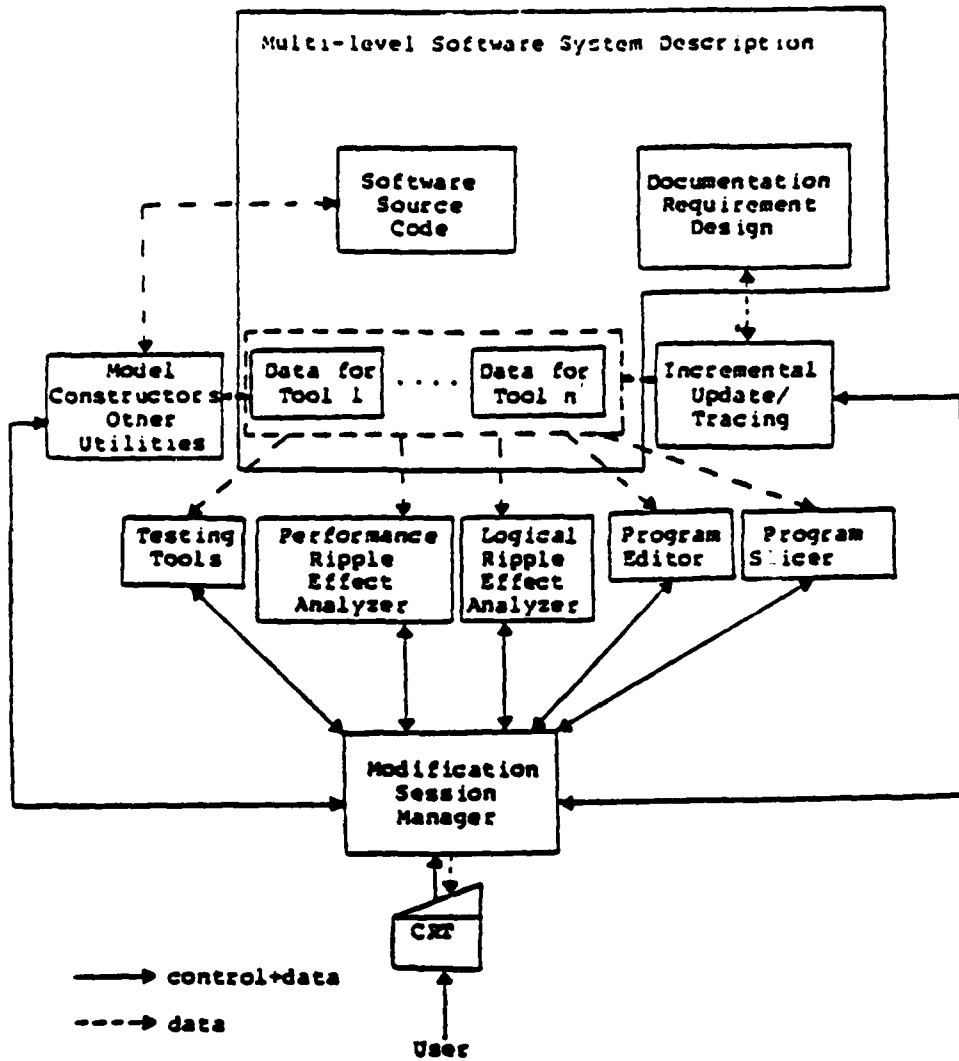
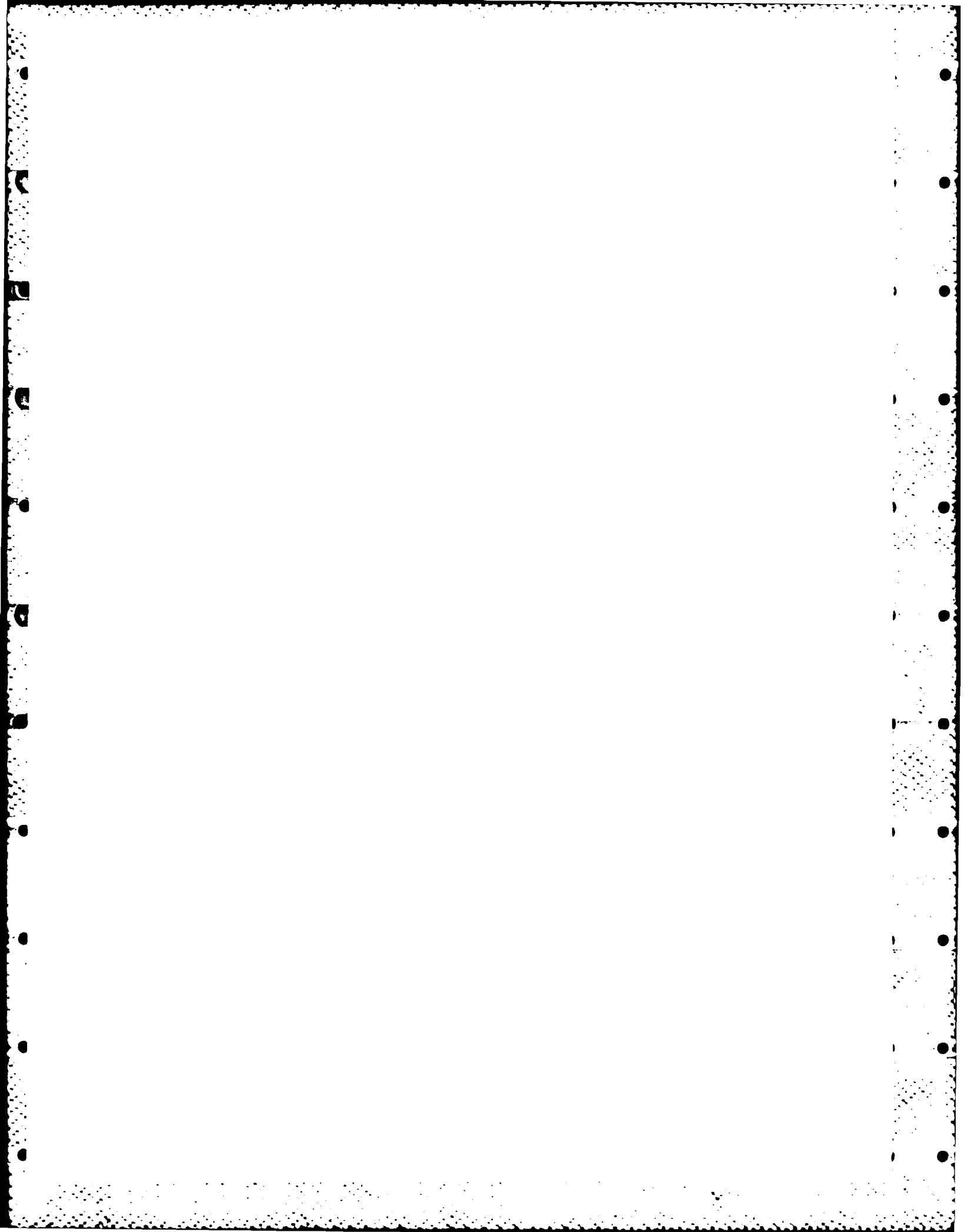


Figure 5. An integrated software maintenance environment



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 (C³I) 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.



FINED

FINED