

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

AD A 126146

RADC-TR-82-313
Final Technical Report
December 1982



ADVANCED TOOLS FOR SOFTWARE MAINTENANCE

Advanced Information & Decision Systems

Jeffery S. Dean and Brian P. McCune



APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED

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

DTIC FILE COPY

02 08 29 031

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

RADC-TR-82-313 has been reviewed and is approved for publication.

APPROVED:



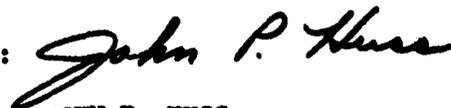
DOUGLAS A. WHITE
Project Engineer

APPROVED:



JOHN J. MARCINIAK, Colonel, USAF
Chief, Command and Control Division

FOR THE COMMANDER:



JOHN P. HUSS
Acting Chief, Plans Office

If your address has changed or if you wish to be removed from the RADC mailing list, or if the addressee is no longer employed by your organization, please notify RADC (COES) 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)

the programming process (the "Programming Manager"), a tool to aid in the collection and use of documentation ("the Documentation Assistant), and an editor that is knowledgeable about what it is editing ("the Intelligent Editor"). The nine tools are based on the computer science technologies of artificial intelligence (particularly knowledge-based and expert systems), automatic programming, intelligent user interfaces, formal verification, software engineering, programming environments, software metrics, and computer-assisted instruction.

Block 19 Cont'd

intelligent user interface
program verification
program annotation
software engineering
programming environment
software testing
software metrics
computer-assisted instruction (CAI)

software tool
management of programming
program editing
style analysis
software change
propagation
tool evaluation.

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE(When Data Entered)

Acknowledgments

The authors thank Prof. David C. Luckham of Stanford University for his contributions to this report, particularly in the areas of ADA tools, ADA style guidelines, and formal annotation and verification. Daniel G. Shapiro and J. Roland Payne of AI&DS provided helpful criticism of the report and contributed to the results reported herein. Other consultants to AI&DS during this project were Prof. Elaine Kant, Carnegie-Mellon University, and Prof. Alan W. Biermann, Duke University.

The Air Force software maintenance problem was studied in part by a series of interviews and questionnaires. We are grateful to the following organizations that participated: Air Force Satellite Control Facility, Sunnyvale Air Force Station, Sunnyvale, California; Data Dynamics, Inc., Mountain View, California; Air Force Communications Computer Programming Center, Tinker Air Force Base, Oklahoma; Strategic Air Command Data Systems Organization, Offutt Air Force Base, Nebraska; and Rome Air Development Center, Griffiss Air Force Base, New York.

Techniques relevant to software maintenance tools are under research and development at a large number of university, government, and private laboratories. Computer science researchers in the following organizations were interviewed during this study: Stanford University, Massachusetts Institute of Technology, Carnegie-Mellon University, Harvard University, University of California at Irvine, and Xerox Palo Alto Research Center.

COPY
REPRODUCED
e

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

Table of Contents

1. Executive Summary	1
1.1 Objectives	1
1.2 Approach	1
1.3 Results	2
1.4 Guide to Reading	3
2. Introduction	4
2.1 Goals and Approaches	4
2.2 Assumptions	5
3. An Introduction to Software Maintenance	7
3.1 What Is Software?	7
3.1.1 The Software Life Cycle	8
3.2 What Is Maintenance?	9
3.2.1 The Software Maintenance Cycle	9
3.3 Why Is Maintenance an Important Problem?	10
3.4 Why Is Maintenance Done?	11
3.5 How Is Maintenance Done?	12
3.5.1 Typical Maintenance Environment	12
3.5.2 Typical Maintenance Administration	13
4. Air Force Software Maintenance	15
4.1 Overview of Selected Air Force Sites	15
4.1.1 Satellite Control Facility	15
4.1.2 Communications Computer Programming Center	19
4.1.3 Strategic Air Command Data Systems Organization	21
4.2 Air Force Software Maintenance Problems	23
4.2.1 Reasons for Software Modification	23
4.2.2 Software Maintenance Functions	24
4.2.3 Software Maintenance Problems	25
4.2.4 Summary and Analysis	25
4.3 Future Air Force Programming Environment	26
4.3.1 The Ada Language	26
4.3.2 The Ada Programming Support Environment (APSE)	27
4.3.3 Ada Programming Methodology	28
4.3.4 The Effect of Ada on Software Maintenance	29
5. The Comprehension Problem	30
5.1 Understanding Software	30
5.2 Documentation	31
5.3 Program Entropy	33
5.3.1 Causes of Entropy	34
5.3.2 Measuring Entropy	35
5.3.3 Controlling Entropy at the Micro-Level	36

5.3.4 Controlling Entropy at the Macro-Level	36
5.4 Summary	37
6. Overview of Software Technology Research Areas	39
6.1 Artificial Intelligence	39
6.2 Automatic Programming	40
6.3 Very High Level Languages	41
6.4 Program Transformation	43
6.5 Formal Verification	43
6.6 Symbolic Execution	45
6.7 Software Metrics	46
7. Review of Related Work	48
7.1 Existing Advanced Programming Environments	48
7.1.1 UNIX	49
7.1.2 INTERLISP	53
7.2 Current Research Projects	56
7.2.1 Stanford Verification Project	56
7.2.2 Programmer's Apprentice	57
7.2.3 Gandalf	59
8. Designing Advanced Tools to Ease Maintenance	62
8.1 Software Productivity	62
8.2 Trends for Advanced Tools	63
8.2.1 Integration - Tools That Work Together	64
8.2.2 Intelligent User Interfaces	65
8.2.3 Domain Knowledge and Reasoning	66
8.2.4 History - Keeping Track of What Has Been Done	67
8.2.5 Incrementalism	68
9. Advanced Tools for Software Maintenance	69
9.1 Programming Manager	71
9.2 Intelligent Editor	78
9.3 Documentation Assistant	83
9.4 Style Analyzer	87
9.5 Metrics Tool Set	91
9.6 Annotation Language	94
9.7 Change Propagation Detector	97
9.8 Test Case Analyzer	100
9.9 Intelligent Tutor	103
10. Evaluating Advanced Software Maintenance Tools	107
10.1 Techniques for Evaluation	107
10.1.1 Evaluating Criteria	108
10.1.2 Evaluating Tools	108
10.2 Tool Ratings - Maintenance Problem Areas	111

10.3 Summary	111
11. Conclusions	113
11.1 Accomplishments	113
11.1.1 Identification of Major Maintenance Problems	113
11.1.2 Identification of Useful Tools	113
11.2 Recommendations	114
11.2.1 Proposed Tools	114
11.2.2 Development Strategy for Proposed Tools	115
11.2.3 Other Research Areas	116
11.2.4 Other Recommendations	117
11.3 Deploying New Tools	118
11.3.1 The Ada Programming Support Environment	118
11.3.2 Retrofitting New Tools to Old Applications	118
11.3.3 Training People to Use New Tools	119
Appendix A. Ada Style Guidelines	121
Appendix B. The Problem Questionnaire	128
Appendix C. The Criteria Evaluation Questionnaire	132
Appendix D. Tool Evaluation Questionnaire - Air Force Evaluators	137
Appendix E. Tool Evaluation Questionnaire - AI&DS Evaluators	143
Appendix F. Tool Evaluation Result Scores	149
References	152

1. EXECUTIVE SUMMARY

This is the final technical report of the Software Maintenance Techniques project conducted for the Rome Air Development Center (RADC) by Advanced Information & Decision Systems (AI&DS). This chapter provides a synopsis of the work performed.

1.1 OBJECTIVES

The cost to the Air Force of maintaining software is measured in the billions of dollars annually. The objectives of this project were to identify, design, and evaluate software maintenance tool and technique concepts (for the Ada Programming Support Environment) that, if implemented, would help increase productivity, improve reliability, and lower costs. The focus was on advanced technologies, such as artificial intelligence and formal verification, that are still considered research topics and are not yet employed in existing programming environments, but which could have major impact if employed in the future.

1.2 APPROACH

Five tasks were performed to achieve the objectives:

1. Software maintenance problem definition: To understand the software maintenance problem, a survey of the literature on software maintenance was performed. Then, extensive interviews were conducted with maintenance programmers and managers at three Air Force C3I sites (the Satellite Control Facility, Communications Computer Programming Center, and Strategic Air Command Data Systems Organization). A questionnaire, designed to assess maintenance problems in more depth, was sent to selected personnel at all the interviewed sites.
2. Tool evaluation criteria: Criteria for evaluating advanced software maintenance tools were defined. These criteria were evaluated by means of a questionnaire sent to the previously mentioned Air Force sites.
3. Advanced technology investigation: Advanced software technologies were investigated by reviewing the literature, attending conferences and other lectures, and making visits to selected research institutions (including Stanford University, MIT, Harvard University, Carnegie-Mellon University, Duke University, and University of California at Irvine).
4. Tool definition: To design advanced maintenance tools, a list of several dozen tools was enumerated. After extensive study and discussion, the list was pared down to nine tools, which were then defined in more detail.

Approach

5. Tool evaluation: To confirm our belief in the desirability and feasibility of these nine tools, they were rated by means of two questionnaires, one for the previously interviewed Air Force software maintainers, and one for the AI&DS project team. The responses were analyzed, based on the evaluation in step two, resulting in a weighted score for each candidate tool.

1.3 RESULTS

Four major problems in the software maintenance process were identified:

- high turnover of personnel and lack of qualified replacements
- understanding what a software system does, without good documentation
- determining all relevant places to make changes
- diagnosing and monitoring operations

These problems (especially the first three) revolve around the issue of comprehension: for maintainers to work effectively with complex software systems, it is necessary for them to comprehend how these systems work.

Many concepts for software maintenance tools were considered. Nine tools to help solve some of the major software maintenance problems were defined. These tools draw upon various research areas, including artificial intelligence, formal verification, automatic programming, intelligent user interfaces, software engineering, software metrics, programming environments, and computer assisted instruction. These tools are described briefly here:

- The Programming Manager assists the programmer by systematically applying administrative and technical policies, as well as helping apply both general and application-specific programming techniques and methods.
- The Intelligent Editor provides facilities for manipulating programs at several conceptual levels (e.g., textual, syntactic, semantic, and intentional), and provides an intelligent interface to other tools.
- The Documentation Assistant is a tightly woven collection of tools for creating, structuring, maintaining, and accessing all forms of documentation.
- The Style Analyzer checks programs for adherence to programming standards and style guidelines (which are expressed with a specification method that is independent of the analysis process itself).

Results

- The Metrics Tool Set provides tools for measuring, analyzing, and assessing various properties of software systems over their lifetime.
- The Annotation Language is a method for extending a programming language by allowing annotations which specify state properties and other aspects of programs that cannot be conveniently expressed in the programming language itself.
- The Change Propagation Detector analyzes a program for effects of a program change.
- The Test Case Analyzer allows the output produced by test runs to be automatically checked for correctness, based on a formal (or informal) specification of what the output should look like.
- The Intelligent Tutor uses a knowledge-based approach to teach programmers about programming languages and programming environments, using the tools themselves.

All of these tools are good candidates for further research and development. However, the evaluation process identified three tools as particularly important for the medium term (3-7 years): the Programming Manager, the Documentation Assistant, and the Intelligent Editor. For short-term payoff (within 3 years), simplified versions of the Documentation Assistant and Metrics Tool Set are recommended.

1.4 GUIDE TO READING

The next chapter discusses the goals of this project, and the approach taken towards reaching those goals. Chapter 3 is an introduction to the software maintenance process. In Chapter 4 the specifics of software maintenance in the Air Force are presented. Chapter 5 explores the main problems of the maintenance process. Chapter 6 presents an overview of several research areas from the fields of artificial intelligence and software engineering that can impact the maintenance process. A review of related work is presented in Chapter 7, including existing programming environments and current research efforts. In Chapter 8, design issues for advanced software maintenance tools are discussed. Chapter 9 proposes nine high level designs for tools to improve the maintenance process. In Chapter 10 these tools are evaluated. Chapter 11 presents the conclusions of this project, including recommendations for tool development and further research on supporting topics.

2. INTRODUCTION

The primary objective of this project has been to identify effective tools and techniques for easing the task of maintaining software systems, assuming a programming environment such as that provided by an Ada Programming Support Environment (APSE). The focus has been on utilizing advanced technologies (e.g., artificial intelligence, formal verification, software metrics), research on which has resulted in techniques that are applicable today or in the near future.

In identifying tools and techniques, this study focused on one aspect of the maintenance problem: specific methods for improving parts of the maintenance process. No attempt was made to provide a "total solution" (if such a goal even makes sense). This approach, specified by the project sponsors, is a pragmatic approach to a significant problem; namely, identifying techniques that can be used today, and techniques that can be used in the future, to ameliorate problems in the arduous process of software maintenance.

2.1 GOALS AND APPROACHES

The goals of this project were specified in the Statement of Work. A restatement of each of the goals, with an explanation of the approach taken in the course of this project, follows:

Goal: Develop criteria and metrics by which tools can be evaluated for their impact on software maintenance problems.

Approach: To create a set of measurements for evaluating tools, the following tasks were performed:

- conducted detailed interviews with all levels of maintenance personnel at three Air Force sites to assess the general problem;
- created a questionnaire designed to probe further into the problems associated with software maintenance;
- created a questionnaire designed to evaluate potential criteria for tool evaluation;
- created two questionnaires for evaluating tools, one for Air Force evaluators (maintainers) and one for AI&DS evaluators (tool developers). The questionnaires were non-overlapping for most evaluation criteria.
- tabulated the results of the questionnaires in order to achieve a rating

Goals and Approaches

of proposed tools.

Goal: Investigate advanced software technologies and evaluate emerging capabilities (based on the above criteria) with respect to their impact on software maintenance problems.

Approach: The approach included reading the technical literature, attending conferences and workshops, conducting in-depth interviews with researchers at a number of universities, and holding many informal discussions at AI&DS. Many technologies were looked at, including artificial intelligence, automated software production technology ("automatic programming"), formal verification, symbolic execution, code optimization, programming aids, very high level languages, program transformations, graphics and other input-output techniques, program annotation, documentation manipulation, software metrics, program analysis, computer assisted instruction, and semi-automated management.

Goal: Develop conceptual designs for maintenance tools that have been identified as having a significant impact on software maintenance problems.

Approach: High level designs for nine tools have been created. These designs include functional descriptions of the proposed tools, a discussion of their interrelationships with existing or planned tools, as well as discussions of why the tool is needed, how it might be built, and further research that may be needed.

2.2 ASSUMPTIONS

A number of assumptions were made during the course of this study:

1. This study focused on Air Force C3I (Command, Control, Communications, and Intelligence) applications. Therefore, the analysis was based on needs particular to this domain. However, there is a great deal of similarity between the observations produced during the course of this study and those reported for other applications; hence, many of the results and conclusions reported here may be considered generally applicable.
2. While the Ada programming language and the accompanying Ada Programming Support Environment (APSE) are not yet in general use, the recommendations in this study have been based on the assumption that this will be the language and environment used in Air Force C3I applications. Once again, these recommendations can be considered generally applicable because Ada and the APSE do appear to provide a solid foundation for software development and maintenance. However, in an environment lacking comparable facilities, the first step towards improving the maintenance

Assumptions

process should be the acquisition of similar facilities.

3. We assume that the organizational environment in which maintenance programmers, analysts, and managers operate will be the same in the future. In particular, neither formal training, duration of assignment, nor management policies will change.
4. Since the goal was to study the application of advanced technologies to software maintenance, little effort was spent looking at existing production-quality, commercially available maintenance tools. The few commercially available tools that were looked into use simplistic approaches that are neither powerful nor flexible. These tools were usually aimed at languages such as Cobol or Fortran, and so some of the features offered by these tools are included as part of Ada.

3. AN INTRODUCTION TO SOFTWARE MAINTENANCE

This chapter provides an introduction to the software maintenance process. It explains some basic terminology and describes the software maintenance process. The focus of this chapter is on the "general" software maintenance process; the following chapter deals with software maintenance from the perspective of the Air Force.

3.1 WHAT IS SOFTWARE?

Some basic definitions are in order:

- Software A realization of a set of plans or specifications, encoded in a computer language.
- Program A piece of software that performs a task or set of tasks. It usually provides limited capabilities, and is not a stand-alone entity (i.e., it requires other software to function properly).
- Documentation Technical exposition and data describing software, including requirements, specifications, and design manuals (including design rationale); code commentary (in-line comments as well as external notes and diagrams); benchmarks; user manuals (and any other information necessary for the operation of the software); and software history (including all changes, results of test runs).
- Software System Traditionally defined as a stand-alone group of programs that provide some set of related functional capabilities. In this report, the definition is extended to include documentation, test data, and possibly special purpose support tools built during the software development stages (e.g., test drivers, debugging aids), as well as the code itself. This extended definition arises from purely practical considerations: it is difficult to operate and maintain a system without this additional material.
- Software Tool A program or software system that assists the programmer in the process of developing and maintaining software.

What Is Software?

3.1.1 The Software Life Cycle

The various phases that a software system passes through during its lifetime is generally referred to as the software life cycle (e.g., [Zelkowitz 78]). The life cycle concept is a descriptive (rather than prescriptive) notion. It divides the lifetime of a system, from initial conception to eventual decommission, into six major phases:

1. requirements analysis: identify the problem (application)

A problem cannot be solved without first identifying what, why, when, and where it is.

2. specification: identify the (software) solution

Once the problem has been identified, the solution can be laid out. The specification phase is an abstract solution, in the sense that it identifies what the solution is, rather than stating details about how the solution works. Since this is a discussion of the software life cycle, the concern here is with the software part of the solution; but note that a specification may also have non-software aspects (e.g., it may include tasks to be performed manually by people).

3. design: describe the software

Deciding how the software works is done in the design phase. Appropriate hardware and software environments are chosen during this phase (unless the problem domain itself has particular hardware and/or software requirements).

4. code: build the software

The software is actually written during the coding phase. The initial stages of debugging (getting the software to work) are also included in this phase.

5. test: ensure the software works

The testing phase is meant to insure that the software performs to specifications. This is not the debugging stage, although bugs may be uncovered. A system enters this phase when the programming group says they have a working system.

6. maintain: modify, adapt, correct, and perfect the software

After the software is finally finished and put into operation, the maintenance phase begins. This phase itself can be broken down into distinct sub-phases (this will be discussed later).

What Is Software?

These steps describe the general process. In any given application, each step may not be precisely delineated (e.g., it may not be possible to identify a time when the system moves from the design phase to the coding phase). Nor are these phases strictly sequential: they may overlap (e.g., certain parts of the code can be tested before the coding is complete), and there are usually feedback loops from some phase to a previous one (e.g., problems during coding may result in redesign; problems found while testing may necessitate recoding). Problems may propagate back several steps; e.g., problems during testing might require a reassessment of the design, the specification, or even the requirements analysis.

3.2 WHAT IS MAINTENANCE?

Software Maintenance is all those activities associated with a software system after the system has been initially defined, developed, deployed, and accepted as operational.

In particular, a maintenance task is usually associated with a request for modification to the software system. These requests are made by users and others affected by the system. This definition takes an inclusive view of maintenance by including enhancement and extension modifications (as well as corrective modifications) as part of maintenance. This differs from the more common exclusive view, which does not consider enhancements and extensions to be part of maintenance. (The notion of inclusive and exclusive definitions of maintenance is due to [Lientz 80]). The reason for using the inclusive definition is based on two observations, which are independent of the type of modification. First, enhancing existing software seems closer in nature to fixing it than developing it. Second, software modifications are more likely to be performed in the maintenance environment (i.e., by maintenance programmers, using tools, documentations, and hardware available to the maintenance team).

3.2.1 The Software Maintenance Cycle

So far, software maintenance has been treated as a single phase in the software life cycle. Looking at maintenance in more detail, it too is a cycle, divided into a number of phases. Each phase of the maintenance cycle corresponds to a phase in the software life cycle. (This leads to the observation that maintenance is really a form of redevelopment.) The steps in the software maintenance cycle are:

1. reanalysis: the problems with the software must first be identified;
2. respecification: then the solution can be identified;
3. redesign: the solution must be mapped out in more detail;

What Is Maintenance?

4. recoding: to implement the solution, old code may have to be changed, and new code may have to be written;
5. retesting: the changed system must once again be tested to ensure that the system satisfies the changed specifications.

Similar to the software life cycle, steps in the maintenance cycle are often imprecisely delineated, and not necessarily sequential. Moreover, some of the initial steps of the maintenance cycle may be bypassed. For example, when modifications are simple, coders may be responsible for implementing changes directly, without a formal reanalysis, respecification, or possibly even redesign phase. Due to the variety of maintenance tasks, this can be advantageous, in terms of cost. However, it is important to ensure that omitting a step will not have a deleterious long term effect on the software system (as might be the case if, for example, major code changes were made without going through a redesign phase). Hence, it is desirable to have some formal method for determining which steps of the maintenance cycle (if any) can be bypassed.

The steps in the maintenance cycle are usually incremental; e.g., there is usually no need to totally redesign the software as a result of modifications, since the original design for most of the system will remain unchanged. The only maintenance step likely to be performed in its entirety is retesting.

3.3 WHY IS MAINTENANCE AN IMPORTANT PROBLEM?

Software costs (including maintenance) in the United States are staggering. Software activities consume three percent of the gross national product. Total software costs in the Department of Defense were nearly \$5 billion by 1979 [Davis 79].

In attempting to reduce overall software costs, maintenance is the first area that should be examined critically. Maintenance comprises a large part of the software life cycle:

Almost every study that attempts to quantize the cost of software maintenance comes up with ball park figures of 50% of total software life cycle costs. Even the most conservative drop to only 40%; and there are some figures at the 80% level. [Glass 79]

Most studies indicate that maintenance costs range from 60% to 75% of total costs.

Why Is Maintenance an Important Problem?

Why does maintenance consume such a large percentage of life cycle costs? While the development phase requires a large initial outlay of resources, the maintenance phase is much longer than the development phase (since maintenance continues for the operational lifetime of the software). The lifetime of a software system is often extended because software tends to evolve over time; maintenance includes enhancements and extensions, as well as fixing things that do not work.

Thus, over the lifetime of a large software system, considerably more time is spent in maintenance than in development. In trying to minimize costs over the total software life cycle, the greatest effect can be achieved by improving and easing the maintenance phase.

3.4 WHY IS MAINTENANCE DONE?

Maintenance is primarily a reactive activity: it is performed in response to requests, rather than on the basis of some regular schedule. Maintenance requests are for modifications to software. Modification requests can be grouped into four basic categories:

1. correcting: "there was something wrong with the system"

A system failure (bug) is found. A failure may be due to nonconformity with specifications (including performance) as well as an implementation problem.

2. adapting: "something the system depended upon has changed"

The operating environment has changed. The operating environment includes both the hardware and (external) software used by the system. Changes of this type are external; they are not intrinsically necessary, in the sense that, if the environment had not changed, the system would not have otherwise required change.

3. perfecting: "we wanted to fine-tune the system"

An enhancement is desired. Perfective modifications may be isomorphic to modifications in any of the above categories. The distinction is that perfective changes are not required to make the system correct. For example, if specifications state that a certain level of performance is required, then the failure to reach that level requires a corrective change; however, if it is decided that "it might be nice" to have the system operate at a higher performance level, then the change is perfective. Such changes may provide improvements in system capability, performance, documentation, etc.

4. modifying: "we didn't like the system the way it was"

Why Is Maintenance Done?

Requirements or specifications are changed. These changes may result from inadequate initial analysis and specification; they may spring from new insights or better ideas about the requirements and specifications; or they may be caused by evolving applications and environments.

This categorization is similar to that described in [Lientz 80], with the exception of the "modifying" request, which was omitted (apparently, it was included under "perfecting").

These four categories can be further abstracted. The first two types of requests, correcting and adapting, are requests for repair: something in the system does not work properly, hence, a change is required. The last two types, perfecting and modifying, are requests for refinement: things were working fine, but some sort of extension or enhancement is desired.

3.5 HOW IS MAINTENANCE DONE?

To improve upon the maintenance process, it is first necessary to understand the non-technical (organizational) aspects: what is the maintenance environment like, and how is it organized? This section presents some generalizations that addresses these questions. The following chapter presents a more detailed view of several selected Air Force sites.

3.5.1 Typical Maintenance Environment

The term maintenance environment is used here to refer to the people responsible for the maintenance task (including programmers, managers, and support staff), as well as the hardware, software, and documentation available to the maintenance staff.

In studying software maintenance, one of the first observations one makes is that maintenance is viewed as a low profile, low prestige task. As a result, the maintenance environment often lacks adequate personnel and resources. System design and development are considered more glamorous, and when a system is complete, the participants often scramble to avoid assignment of maintenance chores. Organizations are forced to staff the maintenance team with junior programmers; as a result, maintenance personnel tend to be less experienced than their colleagues in development.

[Ledbetter 80] has listed a number of other attributes that are fairly representative of contemporary maintenance environments:

- The original system designers and programmers are no longer present.

How Is Maintenance Done?

- In dynamic environments, the original user team is no longer present.
- Many software maintenance personnel maintain a broad span of code (possibly many systems) and are either junior to or inferior to the development team. They also have little or no system design experience and either work alone or as part of a small cadre.
- The test team is no longer present and the test input data and analysis software is out of date. If simulators or drivers were used during the development phase, they are normally useless after a few modifications because they were undocumented and, therefore, basically unmaintainable.
- The original requirements, design, and implementation trade-offs and analysis (performance and capabilities) are not reflected in the system documentation. They were either in project memoranda or communicated orally.
- The requirements, specification, design and interface definition documents have not been kept up to date during the maintenance and modification phase.

3.5.2 Typical Maintenance Administration

The term maintenance administration is used here to refer to the organization and administration of the maintenance environment.

In a preceding section, the steps in the maintenance cycle were outlined. These steps were the technical steps required to carry out the process. The administrative aspects complicate the process. Typically, there is a formal process for making maintenance modification requests: requests made by users must be tracked (until resolved); formal approval is required at many levels, from specification to test acceptance; modified software must be distributed, and users must be notified of changes. Indeed, the whole maintenance process can be obscured (and overwhelmed) by the administrative aspects of maintenance; this leads to the observation that a software maintenance "support system" would be helpful in keeping track of all the steps of the maintenance process.

As an example of the modification request process, listed below are the steps required to make a change in one military environment (reproduced from [TRW 80]):

1. Change Initiation

- receive and process requests
- preliminary analysis and problem/deficiency definition
- preliminary resource allocation and scheduling

How Is Maintenance Done?

2. Change Analysis and Specification

- feasibility
- requirements decomposition/definition
- detailed design
- generate change proposal

3. Engineering Development and Unit Test

- develop the change
- perform engineering tests

4. System Integration and Test

- test system performance
- produce test reports

5. Change Documentation

- document change
- update baseline
- configuration control

6. Certification and Distribution

- certify documentation
- distribute revised data
- provide installation procedures/instructions

This list contains the technical maintenance steps (e.g., develop the change) within the framework of administrative steps (e.g., resource allocation scheduling). Note that the actual changing of code is only a small part of the whole process.

4. AIR FORCE SOFTWARE MAINTENANCE

This study has focused on software maintenance problems of the Air Force. This chapter describes the specific Air Force environments studied, and discusses the problems identified. One of the factors that distinguishes the Air Force environment from industrial environments is the plan to abandon existing outdated programming technologies and jump into a modern programming environment based around the Ada language. This proposed environment is discussed, as well as its effect on the software maintenance process.

4.1 OVERVIEW OF SELECTED AIR FORCE SITES

In gathering data about current Air Force maintenance practices, we were concerned with three aspects: (1) the type of system being maintained (e.g., purpose, scope, duration, size, mode of operation, hardware, languages); (2) the form of software maintenance practiced, in both management and technical aspects; and (3) significant problems encountered. To gather data we visited personnel at three centers of Air Force C3I software maintenance activities: the Satellite Control Facility, the Communications Computer Programming Center, and the Strategic Air Command Data Systems Organization.

4.1.1 Satellite Control Facility

The Air Force Satellite Control Facility (SCF) is located at Sunnyvale Air Force Station, Sunnyvale, California. Its function is to control all Air Force satellites, as well as some of those of other agencies such as NASA and NATO, once the satellites are orbital. The control system's design and implementation, including hardware, software, and communications network, date back to 1967.

Command and control decisions emanate from Sunnyvale and are distributed to eight ground stations located around the world. Ten CDC 3800 mainframes in Sunnyvale are used for trajectory calculations. Varian and Univac minicomputers are used for realtime command and control operations. For each ground station, one minicomputer is located in Sunnyvale and the other at the ground station, with a medium-speed communications channel linking the two. JOVIAL J4 is used for the operating system and most of the applications programs running on the 3800s. All other programs are written in assembly language. There are nearly 1,000,000 lines of J4 code in use.

No programming is done by Air Force personnel at SCF. Maintenance is done by more than ten different contractors. One contractor does system engineering for new requirements; integration, testing, and documentation of the entire system; and "realtime" maintenance of the system during operation. A second contractor maintains the operating system for the 3800s, including utilities.

Overview of Selected Air Force Sites

A third contractor maintains the orbital ephemerous system and a library of standard routines for such things as orbit trajectory calculations that are widely useful to the other contractors. A fourth contractor maintains the realtime software at the ground stations for command and control, data communications, and telemetry processing. Finally, there were seven contractors (at the time of the interview) supporting the software specific to the missions of fourteen special program offices (SPOs). To complicate matters, this last set of contractors is funded by the SPOs themselves, rather than by SCF. Security considerations for some missions make it impossible for the integrating contractor to obtain adequate test data and for the support contractor to know which of his routines are most useful and what new routines would be useful.

A representative contractor is Data Dynamics, Inc. (DDI), which maintains the SCF library of orbit planning and trajectory calculation subroutines. DDI developed the routines over ten years ago and has been maintaining them since. DDI now has a library of 280 J4 routines totaling 120,000 lines of code. DDI suffers from problems such as a slow cycle to have changes approved, difficulty in determining the ramifications of changes to their routines to the software of other contractors, and redundant efforts by other contractors that don't know what routines are already available. For example, without doing a massive and time-consuming search of offline documentation, it is impossible for one contractor to determine what coordinate transformation functions have been written by any other contractor prior to developing a new function from scratch.

Systems Development Corporation (SDC) maintains the operating system for the CDC 3800s. It consists of 300-400 routines written in J4. Most of the other maintenance contractors specialize in programs for dealing with the functions specific to one satellite or class of satellites. Some of the companies involved are General Electric, TRW, Hughes Aircraft, Mellonics, Boeing, and Lockheed.

The only hard data on software maintenance at SCF that we have obtained is the 1980 annual budget for each contractor function and the estimated number of problems reported at each phase in the redevelopment of software to handle a new satellite. Problems include bugs at any level (code, design, or requirements), but not optional requests to alter requirements.

Overview of Selected Air Force Sites

SCF Contractor

Software Maintenance Contract (\$ millions/year)

integration	
system engineering	5.0
integration and testing	5.0
system maintenance	5.0
operating system and utilities	1.3
orbital ephemorous system	1.1
ground stations	4.6
four large missions	5-10 each
ten small missions	1-2 each
TOTAL	52-82

Redevelopment Phase

Approximate Number of Problem Reports

mission development and testing	100
system integration and testing	100
two command exercises	100 each
rehearsal	30-40
fly	200-500
TOTAL	630-940

We were unable to obtain sufficient quantitative data to determine the relative importance of the various software maintenance functions. For example, we would like to have known statistics on who originates a maintenance request (a satellite user agency, SCF, or a maintenance contractor), what type of request it is (requirements change, hardware/software environment change, or bug report), what problems arise in dealing with the request (e.g., can't locate bug, bug appears to be in software of another contractor, etc.), and what the final disposition of the request is. Although all change requests are archived after completion, no statistics are kept. The task of going back through large files of changes manually would be too time-consuming and costly. Without data of this form, we have been forced to base our tool evaluations on qualitative assessments of the problems.

All program maintenance is done using batch job submission and archaic tools (e.g., core dumps). The version of JOVIAL that is now in use, J4, does not allow for separate compilation and loading of modules. And the unstructured nature of the system's COMPOOL makes it difficult to isolate changes. For example, a data structure change that should be transparent to other users may in fact require every contractor to modify his code so that it will continue working. In addition, stringent restrictions on the introduction of permanent changes into the system slow down this process. For these reasons, a new system with formally approved changes is released no more often than once every six months. And a single change requires one and a half years or more to get through the configuration control system. Therefore, when bug fixes or other important modifications to the system are required for the success of a mission, changes to the system's core image(!) are made amidst the normal

Overview of Selected Air Force Sites

satellite controlling operations of SCF.

As mentioned above, there appear to be two types of maintenance activities: bug fixes and formally approved changes. The latter category has two subparts: (1) changes made to fulfill a specific requirement, the largest of which is integrating new subsystems to handle a new satellite, and (2) optional, but perhaps broader changes that redesign an entire subsystem in order to provide more efficient or understandable code. Software maintenance because of changes to the hardware is rare. Maintenance in response to a changing software environment (e.g., operating system or subroutine library) is fully controlled because the entire software environment is being maintained by SCF and its contractors.

Documentation, as with other large, long-lived systems, is a major problem. A lag of two years between changes being implemented and documentation being completely updated is typical. And documentation at SCF costs more than coding.

Finding competent personnel for maintenance programming is difficult. Maintenance programming has always had the stigma of being more mundane than new development. The use of obsolete hardware and programming languages compounds the situation at SCF.

SCF now handles as many as fifty missions concurrently, but a larger capacity will be required by the mid-1980s. Because of this expanded workload, the limited performance of the present system, and the outdated hardware and software, an entirely new satellite control system is now being procured. It will be designed, built, and phased into operation during the next seven years, at a cost of approximately \$200 million. Although this is a large sum, it should be noted that it is equivalent to between only two and four years of maintenance on the present system. With this system, no local processing will be done at the ground stations because they will be connected to SCF via a five megabaud satellite communications link. Program development will be done interactively. ADA will be used as a design language, while the implementation language may be one or more of FORTRAN, JOVIAL S73, and ADA. The plan is to phase into the use of ADA as compilers and other tools become available.

Our interviews with personnel of SCF and its contractors have impressed upon us the importance of the management side of software maintenance, especially when the software system is maintained for multiple computers, missions, and agencies, and by multiple programmers and contractors.

Overview of Selected Air Force Sites

4.1.2 Communications Computer Programming Center

The Air Force Communications Computer Programming Center (CCPC) is located at Tinker Air Force Base, Oklahoma City, Oklahoma. CCPC is the primary Air Force agency for the development and maintenance of communications software, and related software in command and control. Many of CCPC's systems make use of AUTODIN I, the military communications network, as the underlying communications medium. CCPC will also likely be in charge of packet-switched network applications for the Air Force. Among the systems now maintained by CCPC are satellite communications software, emergency communications system (IEMATS), Realtime Autodin Distribution System (RAIDS) for logistics, Strategic Air Command communications system (SACDIN), Automated Message Processing Exchange (AFAMPE), navigational status system, and weather control software.

Unlike systems at SCF, each system maintained by CCPC provides an autonomous function to the Air Force. These systems have been deployed independently over the past fifteen years, typically on hardware that is unique within CCPC and that ranges from minicomputers to large mainframes. The computer was usually chosen by the developing contractor, and 95 percent of the communications software is written in assembly language. The major exception to the use of plain assembly language is the extensive use of COMAC, an assembler macro package in use since 1971 that provides over fifty primitives important to communications programming. This macro package has been ported to a number of different machines. The size of a typical system ranges from 25,000 to 50,000 lines of code, although AFAMPE is 560,000 lines long.

CCPC's maintenance work is done in-house, with no external contractors involved. Hence, control and execution of software maintenance is highly centralized, in contrast to SCF. Because most major new systems are developed via competitive procurement, CCPC's typical mode of operation is to take over a new system at the start of the maintenance phase, i.e., at the time it is first deployed and the developing contractor's work is finished.

CCPC has four relevant directorates: Analysis and Design, Production, Maintenance, and Services. Maintenance handles simple fixes, defined arbitrarily by CCPC to be those requiring under six man-months of effort. Major problems and requirements changes from users go first to Analysis and Design, which does requirements analysis and high-level design. Then that directorate hands the task to Production or Maintenance for detailed design and coding, depending upon whether the task is a major development effort or not. The Services Directorate provides software configuration management of fielded systems, and an independent verification and validation function for both Production and Maintenance.

Maintenance operations are primarily responses to problems, rather than requirements changes. Requirements changes are usually for enhancements to the software. The operating environment (instruction set architecture of the host

Overview of Selected Air Force Sites

computer, and operating system) rarely changes. However, the mix of peripherals on some systems changes frequently. Many problems arise from hardware peripherals and communications software protocols that are changed without prior notification to CCPC.

Most program maintenance is still done using batch job submission and debugging with core dumps, all on the target machine. However, in the case of one recent system that is hosted on a DEC PDP-11, programming is done interactively via CRT display terminals.

Perhaps the biggest maintenance problem at CCPC is the lack of adequate documentation. Documentation standards aren't precise enough, so many contractors deliver poor documentation with new systems. The lack of high-level languages exacerbates the problem. Expending effort on such things as improved documentation wasn't pushed in the past because the time span and cost for maintenance weren't considered any more significant than for development. However, the typical CCPC system was expected to be maintained for eight years and has actually existed for twelve to fourteen years.

Finding, training, and keeping qualified personnel is a major problem for CCPC. With the high demand for analysts and programmers in industry, it is hard to keep good people at CCPC in either uniformed or civilian positions. CCPC is currently staffed at only seventy percent of its authorized manpower level.

The major logistics problem is that of having many remote sites, each with a slightly different hardware configuration that requires different software options. The on-site operators are trained only in the use of the system, not in technical aspects of its maintenance. CCPC is faced with the task of remotely determining whether a problem is due to a hardware, software, or operator error. If it is software, the subsystem at fault must be isolated. This is becoming more difficult as complex software systems supported by different agencies must interact via complicated protocols. Then the situation must be reconstructed at CCPC in order to pinpoint the problem, fix it in all required places in the code, and distribute software updates to the appropriate sites. Often the testbed environment at CCPC doesn't exactly duplicate the hardware/software configuration in the field, and complete information on the status of the system in the field cannot be obtained remotely.

CCPC believes that future software will be written in a high-level language and maintained interactively on a timesharing mainframe. ATECS, a system now being planned that will monitor wideband communications circuits in Europe, will be written in JOVIAL J73. CCPC is currently participating in the testing of the interim J73 compiler. CCPC is also participating in the study and evaluation of the ADA tools being developed by RADC.

Overview of Selected Air Force Sites

From our interviews with personnel at CCPC, we have concluded that a major problem in the maintenance of highly distributed systems (such as systems for communications) is the fast isolation of problems that occur remotely from the central maintenance site. Since the trend toward geographically distributed C3I systems is accelerating, this problem is likely to intensify.

4.1.3 Strategic Air Command Data Systems Organization

The Strategic Air Command Data Systems Organization is part of Headquarters, Strategic Air Command (SAC), located at Offutt Air Force Base, Nebraska. Its mission is to support all of SAC's data processing requirements, which date back to the 1950s, as well as those of a few affiliated organizations. Thus, applications range from business data processing in support of SAC bases to sensor data processing and planning the national strategic war plan.

SAC has twenty-four million lines of code to maintain. Most new systems (including major upgrades) are done by contractors. Outside maintenance contracts amount to approximately twenty million dollars per year. Other maintenance of operational systems is done in-house. Maintenance staffs are organized by functional areas.

We interviewed staff in the four divisions that support C3I systems. These divisions are the Warning Programming Division of the Directorate of Sensor Support, the Force Control Programming Division of the Directorate of Computer Applications, the Intelligence Applications Programming Division of the Directorate of Computer Applications, and the Program Management Division of the Directorate of War Plans Programming. The Warning Programming Division maintains the national strategic warning system used by SAC, North American Air Defense Command (NORAD), and the National Military Command Center (NMCC). The Force Control Programming Division maintains the systems that allow the SAC command to communicate with SAC bases and to keep track of its missile and bomber forces. The Intelligence Applications Programming Division maintains various intelligence data handling systems for dealing with images, electronic signals, and other forms of intelligence information. The Program Management Division oversees the functions of the other, applications divisions within the Directorate of War Plans Programming. This directorate maintains the national strategic war plan, known as the Single Integrated Operational Plan (SIOP).

Since the scope of applications is broad and the number of individual hardware and software systems is large, it is not surprising to find a host of hardware models and programming languages in use. Hardware manufacturers include International Business Machines (and compatible models from other manufacturers), Honeywell, Burroughs, Univac, Digital Equipment Corporation, ROLM, SEL, Perkin-Elmer, Hewlett-Packard, and Data General. Languages used include various versions of JOVIAL (including J3), COBOL, FORTRAN, PL/I, and assembler.

Overview of Selected Air Force Sites

SAC will likely move toward J73 and ADA eventually, but it has been slower than, say, the Satellite Control Facility in this regard. The War Plans Programming Directorate has tried to standardize on PL/I for the past few years, but conversion of old programs hasn't happened. This has left it with all of the old languages still in use (primarily COBOL and FORTRAN), plus another, new language.

The SIOP is a very complex system of programs that serves a central role in the nation's defense. There are over 2.5 million lines of code in over five hundred separate programs. The cycle of a change request to the SIOP is interesting to note. Requirements changes (as opposed to bug fixes) have to be approved by local management and then by a high-level group in the Pentagon. This process is time consuming, yet once approval is given the War Plans Programming Directorate is under considerable pressure to effect the necessary changes as soon as possible. Nevertheless, due to a number of reasons having to do with the complexity of the programs and the importance of the application, a typical SIOP release cycle takes up to five years, with up to two years for software modification. This application is an obvious candidate for new programming languages, design and documentation standards, and advanced maintenance tools.

Changes to the Warning Programming Division's Command and Control Processing Display System must also be approved outside of SAC, in this case by a three-star group from SAC, NORAD, and the NMCC. Compatible versions of this program are maintained at a number of command centers.

SAC estimates that its software maintenance efforts consume even more than the 60-75% of the overall software life-cycle effort found in other places. The maintenance problems at SAC are similar to those found at SCF and CCPC. However, in addition to the problems of documentation and making changes, SAC management was much more vocal about the personnel problems: high turnover and lack of qualified staff. SAC has taken two important steps toward a solution: (1) it relies upon a core of civilian programmers in each division to maintain continuity, and (2) it has received dispensation from the Air Force to allow its uniformed programmers to take a tour of duty averaging more than three years. Both of these steps appear to have helped SAC and are to be highly recommended.

Many systems being maintained lack adequate tools (e.g., even a basic cross-reference listing generator). This forces current maintenance programmers to get by with what they have, or attempt to fit tool building into their already crowded schedules. Unlike CCPC, SAC has no central group of technology watchers or developers that attempts to develop crucial tools or monitor developments outside of SAC that might be transferred inside. This function appears to take place in a very distributed and uncoordinated fashion.

Overview of Selected Air Force Sites

Virtually all of the divisions interviewed suggested the need for procedures and tools for management and coordination of the maintenance process, to ensure that activities (especially documentation and testing) are done by the right person at the right time. Tools might range from simple on-line checklists to fully automated retesting procedures.

SAC could benefit from aids to maintaining continuity of its software maintenance functions, such as aids for training programmers, maintaining quality documentation, and managing the maintenance process. It is also clear that, in the long run, standardizing on ADA as the programming language would help immensely.

4.2 AIR FORCE SOFTWARE MAINTENANCE PROBLEMS

We created a questionnaire to clarify the information learned during the site visits. This section summarizes the results of the questionnaire. The questionnaire and responses are included in Appendix B.

The questionnaire was divided into three parts:

1. reasons for software modifications: "Why is software modified?"
2. software maintenance functions: "Where is time spent during maintenance?"
3. difficulties: "Why is maintenance so hard?"

The results of the questionnaire confirmed the information we collected during the site visits. Due to the diversity of the selected sites, there was (as expected) a good deal of variance between individual responses. The questionnaires were given to a total of nineteen people in the three Air Force Organizations discussed above. There was no attempt to select organizations that were similar or representative.

4.2.1 Reasons for Software Modification

Given the four categories of maintenance requests (section 3.4), respondents were asked to estimate the percentage of requests that fell into each category. The averaged responses, in decreasing order, were as follows:

Air Force Software Maintenance Problems

REQUEST	PERCENTAGE
modifying	46%
correcting	31%
perfecting	15%
adapting	8%

Modification requests alone account for almost half of the maintenance requests. Together with perfecting requests (the other category for refinement requests), they account for over 60% of the maintenance requests (similar to [Lientz 80], where refinement requests were found to account for over 50% of the maintenance requests). Traditionally, maintenance has generally been associated with repairing software. However, these results help confirm the importance of software evolution (i.e., refining, as compared to repairing) as part of the maintenance phase.

4.2.2 Software Maintenance Functions

In this section, respondents were asked to rate the importance (based on amount of time spent) of a number of software maintenance tasks. The scale was an integer from 0 to 10, with 0 signifying "no time spent" and 10 signifying "extreme amounts of time spent". The averaged responses, in decreasing order, were as follows:

TASK	IMPORTANCE
retesting	6.5
recoding	6.3
training of new maintenance personnel and users	4.8
monitoring, problem detection, diagnosis, resolution	4.7
redesign	4.4
redocumentation	3.9
management	3.6
configuration control	3.4
reanalysis and respecification of requirements	2.9

It is interesting to note that more time was spent on lower level tasks (such as testing and coding) than on higher level tasks (such as redesign, configuration control, and respecification). Unfortunately, our questionnaire was not designed to discover the reasons behind this distribution of effort; we cannot tell if lower level tasks were performed more often, or if they were just inherently more time consuming. However, based on general knowledge of the software maintenance environments, we hypothesize that lower level tasks

Air Force Software Maintenance Problems

are actually performed more frequently (and higher level tasks are being somewhat neglected). If this is the case, it will most likely have a negative impact on the overall maintainability of software; e.g., recoding without redesigning or redocumenting can easily result in inadequately documented software that lacks coherent structure.

4.2.3 Software Maintenance Problems

The last section of the questionnaire identifies four major software maintenance problems, and asks respondents to rate the importance of each on a scale from 0 to 10, with 0 signifying "problem is not worth solving" and 10 signifying "it is extremely important to solve the problem". The averaged responses, in decreasing order, were as follows:

PROBLEM	IMPORTANCE
high turnover of personnel	8.7
understanding software (lack of good documentation)	7.5
determining all relevant places to make changes	6.9
monitoring and diagnosing operations	6.3

The major problems all appear to revolve around a lack of knowledge and comprehension of the software. High turnover results in a lack of personnel experienced with the software; lack of good documentation makes it difficult to train personnel; difficulty in deciding all relevant places to make changes is a result of inadequate information and comprehension. Only the last category, difficulty in monitoring operations, really refers to another type of problem, one of a more technical nature ("how can I watch the software system in action to make sure it is working correctly?").

In this section, we also asked respondents to include and rate categories we may have omitted. Of the responses, few were considered as important as the above categories, and those that were tended to be quite environment specific.

4.2.4 Summary and Analysis

The results of the questionnaire shed light on three important issues in the maintenance process. First, most of the requests for maintenance are requests for refinement, rather than requests for repairs. This seems to indicate that maintenance is primarily a process of evolution. Second, most of the time spent in maintenance is spent on low level tasks, such as testing and coding. Not enough information was gathered to assess the importance of this breakdown: Are higher level tasks being neglected, or are lower level inherently more time consuming? Finally, most of the difficulty in the maintenance process seems to

Air Force Software Maintenance Problems

arise from a lack of understanding of the software, which, in turn, is related to the lack of readable, up to date documentation.

4.3 FUTURE AIR FORCE PROGRAMMING ENVIRONMENT

4.3.1 The Ada Language

The Ada language is the ambitious attempt of the Department of Defense to provide a standard programming language. Its use should be expected to have a significant effect on reducing software maintenance costs. Though the language has a number of shortcomings, it will provide a significantly better medium for programming than existing DoD languages (of course, just the acceptance of one language by all the military services is enough to help reduce costs). This section discusses how various Ada qualities will impact the software maintenance process.

Ada - The Positive Side

Ada has a number of important positive qualities, especially for maintenance. These include:

- readability - Ada syntax and notation was designed to be particularly readable. In fact, emphasis was placed on program readability over ease of writing [Ada 80]! The designers were very concerned with visual fidelity; i.e., "how will it look when printed out?" The designers also tried to make use of "English-like" constructs.
- structure - Ada is a highly structured language. It has flexible data structuring facilities: packages provide data abstraction; the with clause allows hierarchical structuring of modules. A rich set of control structures is also provided: the loop construct is more general than the constructs provided by most other languages; exceptions provide a clean way of handling unusual conditions and allow considerable flexibility without resorting to the goto statement.
- generality - Though the Ada effort was focused on embedded software applications (large scale and real-time systems), the language is general purpose, and will be used for many other applications. For non-embedded applications, it is expected that many of the more esoteric features of the language will not be used (e.g., tasks, machine code insertions).
- flexibility - The flexibility and extensibility of the language is impressive: both functions and operators can be overloaded, allowing a natural way for handling similar functions for a diverse set of data objects; generics provide a powerful technique for creating isomorphic objects.

Future Air Force Programming Environment

- protection - Strong typing is an important form of protection; it allows the detection of erroneous constructs before they can even be executed (i.e., at compile time). Yet there is sufficient flexibility in Ada typing so that programmers can do what is necessary without frustration. The designers have also tried to avoid "error prone notations."

Ada - The Negative Side

There is only one negative aspect of Ada that touched upon here. One of the major problems to be faced in using Ada is the complexity of the language. Ada has an overabundance of constructs and rules. For comparison purposes, it is instructive to look at Pascal, a language so successful that all the top proposals for the Ada language used it as their basis.

The success of the programming language PASCAL is in part due to its simplicity. A rough comparison of complexity can be made by comparing documentation of systems. The standard reference guide for Pascal users [Jensen 74] includes both a user manual and a reference manual, yet appears smaller in size than the Ada reference manual alone. Even Pascal has constructs that might confuse users; for example, one computer scientist [Sale 81] has suggested that the repeat/until loop construct in Pascal is too complicated(!), and often misunderstood by programmers. [Sale 81] goes on to suggest that the construct should be removed from the language, arguing that its removal would "...enhance the utility of Pascal by increasing the probability of correctness in programs." While many may not take this suggestion seriously, the question raised cannot be ignored: if people find Pascal constructs difficult to use, what will happen when they start to use Ada? This concern is reinforced by the observation that many computer scientists currently working with Ada also seem to have problems understanding the language.

As a result, training programmers will be a nontrivial task, and they will need much practical experience before they can be considered expert programmers. As suggested above, complexity of a language can have a negative effect on the correctness of programs. It will be particularly important to provide programmers with tools and aids that will help them manage the complexity.

4.3.2 The Ada Programming Support Environment (APSE)

The purpose of an Ada Programming Support Environment (APSE) is to provide support for Ada programs throughout all phases of the software life cycle [Stoneman 80]. Like Ada, an APSE is an ambitious undertaking; its design requirements represent the state-of-the-art in production programming environments. A number of APSEs are planned, each with a set of tools providing a common (minimal) set of capabilities (the MAPSE), built on top of a kernel (the KAPSE) which provides the required, lowest level functionality. As

Future Air Force Programming Environment

a reference point, Unix is an existing environment close in form and function to an APSE (and in fact, could be called an ancestor of the APSE design). All of the tools and technologies proposed in this report could exist quite happily in an environment such as an APSE.

The key features of an APSE are:

- a database, maintaining all the information associated with each project (for its entire life cycle);
- a set of tools, providing the desired support;
- user and system interfaces, providing access to the tools and the database.

The user interface is provided by the APSE command language. In an effort to make the command language natural for Ada programmers, [Stoneman 80] specifies that "...wherever possible, the concepts of the Ada language should be used in the APSE."

The APSE is designed to be easily ported to various machines. It uses a virtual machine approach, and is coded (as much as possible) in Ada.

4.3.3 Ada Programming Methodology

A methodology provides a systematic approach to designing and coding software. Approaches have been based on various techniques, such as the use of data flow or data structures to guide the structuring of the program [Bergland 81].

There is no programming methodology inherent to the APSE. This is unfortunate, since the use of methodologies can help improve the software life cycle process. It has been suggested that the APSE would benefit greatly by taking on more of a methodological orientation, by providing a methodology driven environment [Devlin 81]. The lack of methodology is not altogether surprising, since current knowledge about programming languages has matured more than knowledge about programming methodologies. (It has been suggested that the Ada effort has been somewhat backwards. In a top down approach, the methodology would have been designed first, then the environment, and finally, the language.)

Until methodologies have been selected for the APSE environment, the environment will remain little more than just a set of tools.

Future Air Force Programming Environment

...we will not realize the full potential of Ada until we are able to define a software development methodology complete with management practices which can in turn be supported by automated tools. [Druffel 82]

Work in defining methodologies for the APSE is currently in progress. Hopefully, these efforts will yield results before the APSE has been well entrenched; it is desirable to avoid the phenomenon of having the environment dictate the methodology.

We would make a big mistake in the Ada program if we simply allowed our methodology to be defined by a collection of tools. Certainly there are tools which can be used to support any methodology and which we can begin to build immediately. However, in the longer term we must address the definition of a more complete software environment and better understand where the APSE path leads. [Druffel 82]

Development of methodologies and standards alone is not enough. To ensure that they are applied correctly and consistently, support tools must also be developed.

...programming standards can not be effectively enforced without the aid of automatic tools. Modern programming methodology requires powerful tools to guide and enforce the use of proper practices. [Devlin 81]

4.3.4 The Effect of Ada on Software Maintenance

It is most likely that Ada will dramatically help reduce software life cycle costs. The Ada "movement" has a number of things going for it. First, the introduction of a standard programming language, to be used by all military services for a wide variety of applications, eliminates a large number of problems which arose from the use of many programming languages. Second, Ada is truly a modern programming language; it was heavily influenced by past and current research in programming languages, and it incorporates many features and facilities totally lacking in previous languages. However, on the negative side, the complexity of Ada is of great concern. The complexity issue can be addressed through a number of routes: programmer training, programming standards, support tools, etc. There also appears to be a movement underfoot in the direction of defining subsets of Ada, thus defining sub-languages with sound features but reduced complexity.

5. THE COMPREHENSION PROBLEM

The previous chapters defined software maintenance, and discussed maintenance problems specific to the Air Force. With that as background, this chapter discusses one of the major problems inherent in software maintenance, and suggests some methodologies for reducing the magnitude of this problem.

The difficulty and expense of software maintenance has its root in many factors. Some of these are technical, while other have more of an administrative or political nature. Since the focus of this study was on technical problems and technical solutions, the study does not delve deeply into the administrative issues. Nonetheless, technical issues are not, of course, entirely independent of administrative and political realities.

5.1 UNDERSTANDING SOFTWARE

The first step toward improving the maintenance process is understanding the major bottlenecks or trouble areas. This study has identified the biggest technical problem in software maintenance as one of comprehension. Comprehension refers to the ability to understand how a software system works. To put it simply, software cannot be maintained unless those responsible for maintenance understand the software.

On first impression, changeability (the ease with which code can be modified) is sometimes seen as the major maintenance problem. This perspective is shortsighted. While the final goal of a modification request is to change the software, the process of making the change is only a small part of the entire process. Understanding why a change is necessary, how it should be made, what repercussions it might have, etc., are all important parts of the change process that are not directly related to changeability.

Of the major software maintenance problems identified in the previous chapter, the three (out of four) highest ranked problems are directly related to comprehension. These problems were:

- high turnover of personnel: Experienced personnel are replaced with new personnel who are unfamiliar with the applications software, and may be unfamiliar with the programming tools as well.
- difficulty in understanding (lack of good documentation)
- difficulty in deciding all relevant places to make changes: Programmers have a hard time knowing where to make changes because they do not understand well enough how the code works.

Understanding Software

The comprehension problem goes beyond understanding just the applications software. It includes understanding the bureaucracy for making software changes and comprehension of the maintenance programming environment itself. Programming environments are characterized by some type of physical interface (terminal, card punch), software interface (command language, job control language), and software tools (editor, compiler, debugger). Programming environments, in and of themselves, may be fairly difficult to grasp (just look at the complexity of most commercially available operating systems). To make matters worse, different projects often use different environments. Programmers transferred between projects often have to become familiar with the programming environment as well as the application.

5.2 DOCUMENTATION

One obvious solution to the comprehension problem is documentation. Good documentation dramatically eases the maintenance task by providing information necessary to understand a system. The technique of "just looking at the code" to determine how things work is not sufficient. At best, code specifies what has been done; it cannot specify why things were done, nor can it specify why things were not done. The intent of the original designers and programmers is not discernible from program code.

Documentation includes a wide range of information beyond user and operations manuals. The documentation of a well documented system includes requirements, specifications, design and design rationale, program comments, test data, and results (of test runs), history of changes, and user manuals. Since documentation includes information from every phase of the software life cycle, the task of documenting must be distributed throughout the life cycle. Documentation is not a distinct phase (though some may try to treat it as such).

Unfortunately, documentation is probably the most neglected aspect of the software process. A variety of reasons for this can be cited:

- Unlike other parts of the life cycle, which are discrete steps in the cycle, documentation is continuous. It is an ongoing process, taking place throughout all phases of the software life cycle. Attempts to designate a single documentation phase are misguided, since they fail to take into account the real nature of the documentation process.
- Often, it is not considered part of the deliverable software product; when it is, the specifications for the documentation are usually less rigid than the specifications for the software itself. Consequently, when trying to cut costs and meet tight schedules, documentation is usually the first to slip.
- The usefulness of documentation is not immediately obvious, since it does

Documentation

not actually run (i.e., "it does not do anything").

- It is also much easier to evaluate software than it is to evaluate documentation. Often, it is not until software needs to be changed that the inadequacy of the documentation is recognized.
- Writing understandable documentation is difficult. This problem is compounded when programmers are responsible for all of the documentation, since they often have little training or experience in the area. It is also difficult to judge the quality of the results. Guidelines, formatting conventions, etc., are better developed for code than for documentation.
- Current documentation tools are inadequate; actually, they don't exist in most programming environments. The special needs of documentation are simply not recognized.

Much of the documentation problem is an administrative one: adequate resources must be allocated to the task. Various technical approaches have been tried, none with much success. Some approaches that have been taken are:

- Tools to check programs for adequate documentation (in the form of inline comments) are relatively worthless unless intelligently applied. Using tools of this type to force programmers to comment their code usually results in meaningless documentation.
- Tools to automatically generate documentation from programs are no substitute for human generated documentation. Existing documentation generation tools take a syntactic approach, producing a low level description (possibly improved by incorporating inline comments into the generated documentation).

Technical approaches to easing the documentation problem seem to fall into two basic classes. The first is providing support to the documentation writer. The first tool mentioned above (the comment checker) is an example of this approach. Other existing tools, such as report generators, database systems, program analyzers, etc., can also be adapted, though tools specifically aimed at documentation might be more usable. The other approach is automatic documentation generation. This is a much harder problem. As mentioned above, existing documentation generation tools tend to take a low level approach, producing documentation that reflects the low level nature of the code, rather than the higher level nature of the intentions of the programmer. Also, tools that work off the code are not very useful for other types of documentation, such as requirements and specifications. In the future, sophisticated tools will need to use more intelligent techniques for documentation support and generation.

Documentation

One of the most important steps in the production of adequate documentation is planning. During the planning stages of a software system, documentation goals and milestones need to be developed. Documentation should be given as high a priority as the software. Documentation must be planned carefully. Otherwise, it may become part of the comprehension problem (rather than part of the solution). Documentation generates a need for more documentation. For example, if documentation gets too complex, then the documentation itself must be documented. This "meta" level is necessary for the user to understand the documentation. Unfortunately, it is one more layer that must be comprehended. Also, if the process of creating and modifying documentation is complicated by too many rules and requirements, then the documentation process itself becomes difficult to comprehend.

5.3 PROGRAM ENTROPY

Another factor affecting software comprehensibility is the loss of original program structure and clarity due to changes introduced during the maintenance phase. This phenomenon has been referred to as increasing entropy. The term entropy comes from the physical sciences, referring to the tendency of physical systems to become less organized and less structured over time. The problem has been stated (somewhat dramatically) by Brooks:

All repairs tend to destroy the structure, to increase the entropy and disorder of the system. Less and less effort is spent on fixing original design flaws; more and more is spent on fixing flaws introduced by earlier fixes. As time passes, the system becomes less and less well-ordered. Sooner or later the fixing ceases to gain any ground. Each forward step is matched by a backward one. Although in principle usable forever, the system has worn out as a base for progress... A brand-new, from-the-ground-up redesign is necessary.
[Brooks 75]

The phenomenon of increasing program entropy results in software becoming less comprehensible over its lifetime. The reason behind this is related to the human cognitive processes of trying to understand a complex highly interconnected system. It is necessary to mentally divide a complex system into pieces first, and understand each piece separately. In a highly localized (low entropy) system, the operation of the system as a whole is easily understood on the basis of its component parts. In a less localized (higher entropy) system, the operation of the system cannot be understood solely on the basis of its parts, and so the human must work out another level of understanding based on the interconnection of the segments. People are relatively good at building local models of how systems work; they are not as good at building global models.

Code localization is an important factor in understanding and changing code. A section of code is said to exhibit a high degree of localization if its

Program Entropy

operation can be understood by looking at just it, and no other sections of code. Localization is inversely related to entropy. As the program localization decreases, it becomes harder for the programmer to predict the effects of a change. Changes to one part of a program are more likely to have side effects on other parts of the program. With an inadequate model of the program, the programmer may not notice these propagated side effects, resulting in program errors. This then requires changes to changes, resulting in a "cycle of changes."

Not only does increasing entropy make it more difficult to understand how the system works; it also becomes more difficult to explain how the system works. As a result, documentation suffers, even if an attempt is made to keep it up to date. The net effect of increasing entropy is an increase of effort necessary for maintaining software. [Belady 71] has claimed that the increase in entropy has an exponential effect on total effort required to maintain a system.

5.3.1 Causes of Entropy

Software maintenance is inherently an entropy increasing process. Unless work is done specifically to decrease entropy, the entropy of a software system will tend to increase over time. A number of factors contribute to this behavior:

- monotonic program growth - Programs become functionally more complex over their lifetime. Features are always added but rarely taken away.
- poorly integrated changes - Maintenance personnel are not always familiar with the original design philosophy and decisions, so their changes are inconsistent with respect to the rest of the system.
- lack of time - Due to the reactive nature of maintenance, tasks are often done in the quickest way possible.
- poorly designed program - Program modifications may be difficult to make because the program structure is inflexible and inadequate.
- changes to changes - Failure to design changes that match the complexity of the program and the problem result in a cycle of changes [Glass 81].

There are numerous ways to reduce the problem of entropy increase. Before entropy can be controlled, however, it is necessary to develop some techniques for quantifying entropy.

Program Entropy

5.3.2 Measuring Entropy

Unfortunately, there is no single simple way of measuring program entropy. The real problem lies in developing a concrete, generally accepted definition of what entropy really is. However, based on the intuitive and abstract notion of entropy as "degree of structure," a number of techniques are available or can be developed to give some sort of approximation. An entropy metric is a tool that uses some technique to measure the entropy of a program. If one metric is applied to a program at various intervals during the program's lifetime, the change in program entropy can be observed. Though program entropy tends to increase over time, it does not necessarily increase with each change. It is possible for a change to be entropy preserving, or even entropy decreasing. Note that the net change in entropy due to a modification depends on the particular entropy metric as well as the modification itself.

Entropy is a measure of disorder. Disorder can arise at many levels, from the "macro" level (modules, subsystems, etc.) to the "micro" level (program statements). Macro-level characteristics are dealt with primarily during specification and design phases of the software life cycle; micro-level characteristics are dealt with during the coding and debugging phases. It is useful to make the distinction between macro-level and micro-level entropy because the levels correspond to different life cycle phases. Different metrics can be used to measure and control entropy at different levels. However, this does not mean that there are two kinds of entropy; entropy is entropy, regardless of how it is measured.

A good approximation of micro-level software entropy can be based on a program connectivity metric [Yau 80]. Connectivity refers to the interdependence between program parts (where dependencies can be based on either data flow or control flow). Program connectivity is primarily a micro level measurement; it is based on a graph theoretic analysis of individual program statements. Connectivity as a measure of entropy is consistent with the intuitive notion that an increase in the number of connections between different parts of a software system lead to an increase in entropy. A corresponding macro-level measurement might be logical connectivity, based instead on an analysis of logical (abstract) structure (rather than physical structure).

Entropy metrics are still considered a research area, especially macro-level metrics. While micro-level entropy metrics could be constructed from current research prototypes (e.g., the previously mentioned program connectivity metric), macro-level metrics are still not very well understood. Similarly, techniques for controlling macro-level entropy are similarly not well understood.

An entropy metric produces a single measurement representing a point in some space, which is a static measurement of entropy at a particular time. However,

Program Entropy

maintenance is concerned with the dynamic process, i.e., the change in software entropy over some period of time. Making effective use of entropy metrics requires that measurements be taken at various points in the life cycle of a program, allowing establishment of reference points for future comparisons, and allowing a more critical examination of program evolution.

5.3.3 Controlling Entropy at the Micro-Level

At the micro level, entropy increase can be controlled incrementally by minimizing the entropy increase for each set of modifications. For example, if a program needs to be changed, one can choose among alternative changes by determining the effect of each change on the program's entropy, selecting the change which results in the lowest overall entropy (balanced against other criteria, such as efficiency). Note that it is possible for individual changes to be entropy decreasing; but over the long term, changes tend to result in a net gain.

The impact of micro-level control is not easily predictable. The incremental approach is definitely myopic, looking at the immediate benefit of a particular change, instead of looking at the software as a whole. Macro-level control is more appropriate for long term results. However, micro-level control is still likely to be useful on a more practical and immediate basis. Other techniques can also be used to control micro-level entropy, such as the modern programming practices of modularity and data abstraction. These may have a positive impact even if the rest of the system does not make use of such techniques.

5.3.4 Controlling Entropy at the Macro-Level

As mentioned earlier, macro-level entropy is hard to quantify. Techniques to control macro-level entropy are also difficult to prescribe. The best method for managing macro-level entropy is planning. Long term needs of a software system as a whole need to be articulated during planning. Both the growth and the structure of the system need to be taken into account. These factors are ignored at the micro-level. Some general guidelines can be stated:

- Long term planning can help identify future directions for a software system, allowing designers to build software with sufficient "flex" where necessary. The result is software that is easier to modify.
- The promulgation and enforcement of standards, throughout all phases of development, is an important factor in improving maintenance. Judiciously chosen standards result in better programs, as well as making comprehension easier for the programmer, since things will always be done in standard ways.
- Testing needs and capabilities should be planned from the start. Testability can and should be built into systems. It is difficult to test

Program Entropy

systems with no provisions for testing; they usually require special procedures or tools to be devised. It is also important to ensure that adequate resources are available for testing.

- Also important for starting the maintenance phase on a "good foot" are acceptance reviews and audits. Viewed as an assembly line, software should not be allowed to proceed to the next step of the line without meeting the acceptance criteria for the current step.

Preventative maintenance (PM) for software is also useful for decreasing entropy. Preventative maintenance is a technique generally used in hardware maintenance: Hardware is inspected, cleaned, and adjusted on a regular basis, regardless of whether it is broken. The purpose of preventative maintenance is to help reduce the likelihood of hardware failures. In the case of software, the goal of preventative maintenance is somewhat different: it aims to ease the task of making software modifications (when modifications are necessary). During software preventative maintenance, software is modified to reduce complexity. At least one experiment has shown that preventative maintenance can aid program readability [Elshoff 82].

It is rather unfortunate that preventative maintenance is generally not considered applicable to software. The rationale usually given is that, since software is a logical entity, it will not break unless someone does something to break it; and until that point, there is no need to do anything. In practice, this results in a demand driven process, where things get done only when they no longer work properly; at this point, pressure often makes it difficult to perform a quality repair. But preventative maintenance is justifiable: As long as a software system is operational, it will need to be modified. Spending a small amount of time improving code can save a large amount of time later when trying to modify the code. Incremental approaches to preventative maintenance are possible: when code needs to be changed, some time is also spent improving the code. Preventative maintenance can be used to reduce both micro- and macro-level entropy.

5.4 SUMMARY

The problem of comprehension must be addressed by any tools or techniques that aim at significantly reducing the software maintenance effort. The solutions presented in this chapter are not meant to be comprehensive; they were simply meant to be indicative of what might be done. A later chapter in this report presents some ideas for tools that handle these problems.

Various administrative issues have been ignored here, since the goal of this report is to study technical problems of software maintenance. However, even technical solutions are applied in some sort of administrative environment, and thus, administrative issues should not be ignored. Important administrative issues include: putting design team members on the maintenance team, requiring

Summary

that documentation be given high priority, allocating adequate time and resources for training new personnel, etc. This list could go on and on. Many of these issues turn out to be specific to the particular site in question. For a technical solution to be successful, it must address a problem in the specific environment, as well as meshing with the administrative policies.

6. OVERVIEW OF SOFTWARE TECHNOLOGY RESEARCH AREAS

One of the goals of this project was to investigate advanced technologies which might help improve the software maintenance process. The major topics studied (and discussed in this chapter) are artificial intelligence, automatic programming, very high level languages, program transformation, formal program verification, symbolic program execution, and software metrics. Other areas considered include advanced code generation and optimization, programming aids, graphics and advanced forms of input-output, and computer-assisted instruction.

Future generations of programming tools and systems may well apply many of these technologies. The discussion in this chapter is focused on technologies and research areas in general. Specific uses of some of these techniques are discussed in a later chapter.

6.1 ARTIFICIAL INTELLIGENCE

Artificial Intelligence (AI) is concerned with designing computer systems that exhibit qualities or abilities generally associated with human intelligence, such as reasoning, learning, problem solving, vision, and the understanding and generation of natural language. The field of AI, as we know it today, has been around for roughly twenty-five years; it sprang from the union of the first modern (stored program) computer systems with theories of mathematical logic and computation (due to Turing, Whitehead, Russell, Church, Godel, and others) and the theories of cybernetics and self-organizing systems (due to Wiener, McCulloch, Shannon, and others). The main characteristics of AI systems are symbolic processing (as compared to numeric or textual processing) and the use of heuristics ("rule of thumb" techniques instead of "guaranteed" algorithms). Research in AI has not been limited to any single aspect of human activity; researchers have studied a wide range of problems, including game playing, image interpretation, medical diagnosis, design automation, speech recognition, natural language understanding, cognitive modelling, information retrieval, etc.

Some AI research has focused "inward," applying intelligence and understanding to the programming process itself. These topics, including automatic programming, formal program verification, and intelligent user interfaces, are of direct interest to the software maintenance process. The following discussion of AI tools and techniques will be limited to those applicable to the software development and maintenance process.

The idea of using AI to approach programming has a great deal of intuitive appeal. AI systems are generally realized through their implementation as computer systems; why not apply AI to computer systems? Even a brief encounter with programming makes it clear how fruitful this could be. With current

Artificial Intelligence

programming practices, almost all high level work involved in the software process (e.g., designing programs, writing code, finding bugs) is still the responsibility of the programmer. Tools available in current programming support environments provide little more than "clerical" assistance. For example, a text editor, used to create and modify programs, knows nothing about programs or programming languages; a compiler translates statements (made in a procedural language) to machine instructions, knowing nothing about what the program is supposed to do; a debugger provides for the display and manipulation of data and control flow, but it does not have any notion of what it means for a program to be bug-free. These tools do not reduce the "cognitive effort" required by the programmer to create, fix, or extend a program. They have no knowledge of what the programmer wants to do, is doing, or should be doing.

The techniques and methods of AI address these issues. Reasoning, learning, and problem solving are precisely the things lacking in current programming support environments. The key to the future of programming is intelligence.

6.2 AUTOMATIC PROGRAMMING

Automatic programming is an area of AI concerned with automating parts of the programming process. The notion of automatic programming has changed over the years; the first compilers (written in the 1950s) were called "automatic programming systems" by their designers. The current notion of automatic programming goes beyond high level languages: it deals with specifications.

Automatic programming systems can be identified by four characteristics [Barr 82]:

1. specification method - The programmer must communicate his intent to the computer. Automatic programming systems have provided several methods for specification. One is formal specification, which involves a rigorous specification of the program. This specification usually talks about how the program is to behave (but not how that behavior is to be achieved). Another method is specification by example: Given a number of input/output pairs, the automatic programming system attempts to infer the user's intentions. Yet another specification method is natural language. The programmer simply states "what the program should do" in his native tongue.
2. target language - The target language is the language in which the automatic programming system generates the program. Automatic programming systems generally write programs in a higher level language (such as Lisp or Fortran) which can then be compiled by an existing compiler.
3. problem area - The techniques used by automatic programming systems usually limit these systems to certain types of problems. One technique

Automatic Programming

to increase the power of an automatic programming system is to incorporate knowledge about the problem area; this is especially useful in a natural language specification system, where it is necessary to have additional information to disambiguate the specification and resolve inconsistencies. Some automatic programming techniques are more suitable to specific problems (e.g., mathematical problems, data processing problems).

4. method of operation - Several approaches have been used, with varying degrees of success. These include theorem proving (a formal mathematical approach), program transformation (where a specification is refined by a series of predefined transformations), knowledge-based/expert system (which uses a set of rules that codify knowledge about programming), and "traditional" problem solving (which uses goal-directed searches and heuristics).

Efforts to construct automatic programming systems are still considered research projects. Some of the techniques may begin to make their way into real programming systems, e.g., techniques for translating very high level languages and program transformations.

6.3 VERY HIGH LEVEL LANGUAGES

A very high level language (VHLL) is a programming language that provides capabilities significantly beyond the capabilities offered by traditional high level languages. The "level" of a language refers to its similarity or closeness to machine language. Assembly language is a low level language; it maps directly into machine language, and requires the programmer to be familiar with the basic operations of the target machine. Languages like Fortran and Pascal are considered high level programming languages (HLLs); they provide the programmer with a computational model that is somewhat higher than machine level (e.g., by allowing the programmer to talk about variables and loops, instead of memory locations and jumps). Languages such as APL and Lisp are considered even higher level, falling somewhere in between HLLs and VHLLs; they allow the programmer to talk about arrays, lists, and the composition of operators. VHLLs such as SETL [Dewar ??] provide yet a higher level computational model. VHLLs eliminate much of the need to specify the minute details of how things are done. They may provide non-procedural or non-deterministic structures, which allow the programmer to concentrate on specifying the behavior of the program (instead of worrying about how that behavior is to be achieved). They provide sophisticated data structures, such as sets, infinite structures [Ashcroft 77], etc.

VHLLs are closely related to automatic programming. It can be hard to define the difference between a VHLL and a specification language. VHLLs remain a research topic because the process of translating a VHLL program into an efficient program is difficult. Much work has been done on translating VHLL programs into more efficient programs (e.g., [Schwartz 75], [Low 74], [Dewar

Very High Level Languages

79], [Kant 79]). Lacking efficient methods for translation, VHLLs can still be effectively employed, by virtue of their ability to reduce manpower costs. Even for "semi" VHLLs (like APL), program development times (when compared to languages such as Fortran or Pascal) are often more efficient in terms of programmer time by a factor of ten. "True" VHLLs could reduce that programming effort further. The inefficiencies of a VHLL program can often be more than compensated for by the reduced development and maintenance costs, especially as the price of computers falls and the cost of programming (programmers) rises.

The use of VHLLs in real programming environments is somewhat controversial, and the controversy goes beyond issues of efficiency. Much of the extra "legwork" required in languages like Ada or Pascal (e.g., exhaustive type declarations) is considered a desirable feature by the designers of those languages. If programmers are forced to clearly state the relationships between all parts of a program, there will be less likelihood of error (or so the theory goes). The opposing view is that these features are overly restrictive and that their absence makes the task of programming both easier and more fluid.

Complete specification does allow certain types of program errors to be caught at compile time, with relatively little effort. For example, an Ada compiler will catch a statement that tries to assign the value of variable of type X to a variable of type Y. (Ada does strict type checking.) Contrast this with a language such as APL, which does not do type checking: It even allows a variable to change types during the course of computation. The only time APL will complain is when the type of a variable is inconsistent with its context (e.g., applying an arithmetic operator to character strings). If objects have different but consistent types, APL will perform the necessary conversions (e.g., converting from integer to floating point or converting from integer to boolean). Thus, APL programmers never have to declare variables, and they don't have to worry about type constraints. The issue boils down to ease of use versus rigid specification. For applications such as rapid prototyping, ease of use is more important; for large, long-term projects, rigid specification is more important.

While VHLLs are unlikely to immediately replace HLLs in production programming environments, it may be possible to integrate VHLLs into HLL environments. For example, a VHLL might be incorporated into an APSE to provide a tool for rapid prototyping or to write code that is less critical. There might be some tools (such as program transformation tools) that could, if needed, help convert these VHLL programs into HLL programs. An alternative to a VHLL is a set of library packages that provide very high level functions. For example, infinite objects could be emulated used data abstraction capabilities provided by Ada. These packages might then be placed in a library for general use.

Program Transformation

6.4 PROGRAM TRANSFORMATION

Program transformation is the conversion of a program into another computationally "similar" program, where the degree of similarity ranges from "analogous" to "equivalent." Transformations may be done for a variety of reasons: If a program library contains a routine similar to what the programmer needs, it may be possible to automatically transform that routine into the desired one (e.g., transforming a hashing routine to use chaining instead of open addressing [Knuth 73]); if a program is written in a non-procedural specification language, it may be necessary to transform the program into a more procedural form before it can be translated into some real programming language; if the program is written using inherently inefficient constructs, transformation can convert those constructs into more efficient ones.

Taking the idea of transformation one step further, the entire programming process can be thought of as a series of program transformations or refinements, going from a high level specification to the actual code. In recent years, this concept has gained a good deal of popularity, and has been advocated as the basis for the program development process; for example:

The path from design to implementation should be by a repertoire of transformations, which convert the design into an efficiently runnable form while preserving the correctness of the design. Ideally, these transformations should be mechanized to guarantee the preservation of correctness. [Jackson 82]

Program transformation techniques have potential for short term impact on programming systems. Because of the incremental nature of transformations, it is possible to incorporate new transformations selectively, choosing ones that are tried and tested. Many optimizations performed by compilers can be thought of program transformations which are applied for efficiency (e.g., [Loveman 77], [Arsac 79]). Complex languages (such as Ada) may find particular benefit in transformation techniques as a means of mapping complex constructs into simpler ones whenever possible.

6.5 FORMAL VERIFICATION

Formal verification is the demonstration that a piece of program is consistent with a given specification. This demonstration is carried out as a proof within the framework of a formal system which in most cases is based on first-order predicate logic. The specification formally describes desired properties of the program; it may give a complete specification of functional behavior (relationship between input and output values), or a specification of certain aspects, like absence of particular runtime errors, security of data flows, termination, or bounds on running time.

Formal Verification

Formal program verification requires the following:

- a formalization of the semantics of the programming language under consideration;
- a specification or annotation language, in which a programmer can formally express the concepts that are to be used in specifying properties to be verified;
- a mapping of specifications and annotated programs into theorems which, when proved, express consistency of program and specification; these theorems are commonly called "verification conditions";
- the capability for mechanical assistance in carrying out the required formal proofs as they tend to be rather tedious and lengthy.

Formal program verification is one form of program validation. It differs from others by requiring rigorous and formal specifications as well as the capability for reasoning about programs, and in turn provides a much higher degree of assurance that a program indeed performs as specified. Because of the inherent difficulty and cost in resources involved in formal verification, it should be expected to complement other debugging techniques, rather than replace them.

Research in program verification has led to a variety of techniques and implemented systems (e.g., [Boyer 78], [Good 79], [Gerhart 80], [SVG 79], [Robinson 79]). Most approaches are based on the technique of inductive assertions: A program is augmented by assertions stating properties of input and output variables and certain critical internal states. An axiomatic semantics of the programming language allows the programmer to formally analyze such annotated programs (in particular, to reason about their consistency).

Axiomatic semantics and verification condition generators have been developed for various programming languages, e.g., Pascal [Hoare 73], [SVG 79], Euclid [London 78], Jovial [Elspas 80], and Fortran [Boyer 80]. The programming languages are complemented by specification languages in which the intended behavior of programs is expressed.

Existing verification systems are largely experimental. They have served to demonstrate their usefulness in rigorous verification on at least medium-sized programs. With a fair amount of effort, substantial programs can be specified and verified (e.g., the major parts of compiler for Pascal-like language [Polak 80], and, most recently, a software system for flight control [Melliard-Smith 82]).

Formal Verification

The experience with existing systems has also pointed out certain limitations and weaknesses of already available verification techniques and implementations. Current research is concerned with extending available verification techniques and making verification more widely useable.

Formal program verification is often perceived as expensive and resource-consuming. Available verification systems almost always require an expert user. Current research is to a large extent motivated by the needs of application areas in which the requirement of high reliability justifies the cost of applying formal methods. These include systems or system components that are required to be secure or ultra-reliable (e.g. operating systems kernel, flight-control software).

6.6 SYMBOLIC EXECUTION

Symbolic execution means evaluation of a program with (possibly) symbolic values instead of actual data. Symbolic execution creates symbolic expressions that represent the values of outputs as a function of input variables, and (symbolic) predicates ("path conditions") that characterize the subset of values that cause the program to execute a particular program path. Symbolic evaluation thus shows the dependencies between the values of different variables and between data and control flows.

Symbolic execution provides a versatile and powerful tool for debugging and analysis of programs. In comparison with ordinary testing, one symbolic execution run of a program may correspond to a potentially large (even infinite) number of normal test runs. There are indications that symbolic testing is more effective in catching errors than normal testing [Howden 77].

Symbolic execution is normally done by interpreting the source code; it is thus more expensive than ordinary testing which uses compiled code. Testing by symbolic execution is therefore probably most useful only for fairly small program pieces. On the other hand, a program can be executed symbolically even before all subprogram bodies have been supplied if the programmer provides a symbolic description (specification) of the effect of subprograms which can be simulated by the symbolic evaluator. This mode of testing fits nicely into the currently advocated paradigm of programming with levels of abstractions.

Symbolic execution may be considered a weak form of program verification; it shares some of the problems of verification systems. In fact, some verification systems are really symbolic interpreters with an added facility to understand symbolic assertions, thus providing a capability for program verification (e.g. [King 75], [Boyer 75]). Symbolic execution can normally be performed without programmer-supplied specifications (assertions) and often requires only limited automatic deduction capabilities. The distinction between verifiers and systems for symbolic execution is primarily one of

Symbolic Execution

engineering, depending on what particular capabilities are supported by an implementation. Systems for symbolic execution typically emphasize interactive and more flexible use than systems for rigorous program verification.

Symbolic execution may be used to generate predicates which characterize subsets of the space of input data. These predicates provide a basis for the generation of test cases and selection of test data for which the compiled code is then executed (e.g. [Clarke 76]).

Besides pointing out design and programming errors, symbolic execution may be helpful in identifying possible places of code improvement, for instance by locating dead branches in the code [King 81].

More recent work has been concerned with developing a symbolic evaluator as a kernel for an integrated program development system. A symbolic evaluator builds up a data structure that records symbolically dependencies between variables and control paths for the whole program (as opposed to individual execution runs), thus representing all possible program executions; this data structure is then used by various other tools in the program development system (e.g., [Cheatham 79a], [Cheatham 79b], [Asirelli 79]).

6.7 SOFTWARE METRICS

Software metrics provide quantitative, and thereby hopefully objective measures of program complexity. An example metric is the degree of interconnectivity of a set of modules as determined by an analysis of their data and control flow graphs. These measures can be used to predict estimated development or maintenance effort; to guide the development and maintenance process; or to predict the reliability (lack of errors) of a program. Current research efforts in software metrics grew out of the work done by Halstead (e.g., [Halstead 77]).

No one metric will satisfy all needs; different metrics measure different qualities of a program, and different researchers have different ideas about what is worth measuring. Many different metrics have been proposed and studied; e.g., see [Baker 80] for a comparison of several metrics. Metrics are evaluated empirically, in experiments that apply one or several metrics to a series of programs. These results are then evaluated on a subjective basis.

The uses of software metrics are diverse; they include:

- analysis - Software can be analyzed for "quality," where a high quality program is one with low complexity. This analysis might be used as part of the acceptance criteria for a software system, or might be used to

Software Metrics

assess programmer skill.

- prediction - The complexity of a software system can be used to predict the maintenance effort (where effort might be cost, programmer time, etc.). For example, if a program bug is found in a very complex module, the cost of fixing that bug will be relatively high.
- control - Metrics can be used as a basis for control and decision making in the development and maintenance processes. For example, if there were several ways to correct a bug, the method which increased complexity the least could be selected. Alternatively, if a proposed change resulted in a large complexity increase, the programmer/analyst might be required to find another way of making the change.

There are at least two requirements for the effective use of software metrics. First, reliable metrics must be chosen or created. Reliability means that a metric really measures what it is supposed to measure; it implies that the metric performs consistently over all types of software. Second, baselines may need to be established to interpret the results of applying metrics to software: When metrics are used as an absolute measure (as in the above discussions on analysis and prediction), interpretation is dependent on some global understanding of what the metric means. This understanding can be established empirically by deriving a set of baselines, which compare measured complexity with actual maintenance effort (for some set of maintenance tasks). On the other hand, if metrics are used comparatively (as in the above discussion on control), baselines may not be necessary.

7. REVIEW OF RELATED WORK

This chapter discusses some specific programming support environments and research prototypes that were reviewed in the course of this project.

7.1 EXISTING ADVANCED PROGRAMMING ENVIRONMENTS

A programming support environment is an interactive software system that aids the programmer in many phases of the software life cycle. As one might expect, it provides tools to help the programmer perform common tasks (e.g., editors, compilers, debuggers); it may also provide relatively advanced or unique tools (e.g., specification tools, configuration tools). However, there are a number of features that distinguish a programming support environment from a "plain" programming environment. These have been summarized by [Osterweil 81]:

- breadth of scope and applicability
- user friendliness
- reusability of internal components
- tight integration of capabilities
- use of a central information repository.

The key idea behind a programming support environment is to aid the programmer in all his activities, by providing both tools and tool-building facilities (to construct new tools, as needed), as well as providing a flexible and easy to use ("friendly") interface.

This section describes two of the oldest and best known programming support environments, Unix and Interlisp. Neither Unix nor Interlisp was designed as a total programming support environment. Both started out in life as experimental projects of a much simpler nature: Unix as an operating system, Interlisp as a Lisp interpreter. Over the past decade, they evolved into fairly sophisticated programming support environments in response to user feedback and developer insight.

The discussion overviews the most salient features of each system (but does not attempt to critically evaluate or cover them in detail). For both systems, the discussion covers the basic design philosophy ("why is this system the way it is?"), the key characteristics of each environment ("what distinguishes this system?"), and tools that are particularly interesting.

Existing Advanced Programming Environments

7.1.1 UNIX

Unix [Kernighan 81] is a total programming support environment -- it is an operating system and a set of tools and languages. Unix was developed at Bell Labs in 1969 for a PDP-7 minicomputer. Since then, it has been moved to a variety of different machines, and there are currently several thousand Unix installations around the world.

Philosophy

The Unix design philosophy has been a major factor in the success of Unix. The philosophy has been stated (retrospectively) in [McIlroy 78]:

- Make each program do one thing well. To do a new job, build afresh rather than complicate old programs by adding new features.
- Expect the output of every program to become the input to another. Don't clutter output with extraneous information, and avoid stringently columnar or binary input formats. Don't insist on interactive input.
- Design and build software to be tried early. Don't hesitate to throw away the clumsy parts and rebuild them.
- Use tools in preference to unskilled help to lighten a programming task, even if you have to detour to build the tools and expect to throw some of them out after you've finished using them.

Key characteristics

As a result of the Unix philosophy, Unix tools have tended to be small and conceptually simple, thus facilitating a "toolkit" approach to systems development. The toolkit approach has been enhanced by the Unix command language interpreter, which provides high level "connectives" for putting things together.

Unix was built around the C language. Most of the Unix system is coded in C, and all the facilities of the system are available to the C programmer. In addition to making the code easier to read (and hence maintain), its dependence on C (instead of assembly language) has made Unix a relatively "portable" operating system.

Unix supports a uniform I/O structure, based on a hierarchical file system. File, device, and inter-process I/O are compatible; there is no need to be concerned with the actual source or destination of I/O operations. Files are unstructured and randomly addressable.

Existing Advanced Programming Environments

Tools

This section discusses some of the more notable Unix tools. We include the Unix operating system interface (the Shell) here because it really is a Unix tool (unlike other computer systems, where the operating system interface is a special program).

Shell

The Shell [Bourne 78] is a command programming language that provides an interface to the Unix operating system; it is the basis of the Unix programming support environment. A number of different versions of the Shell have emerged over the years. Most have a large number of features in common; the features we describe here should be available in every Shell (in some form or another).

As a language interpreter, the Shell provides control and data structures. For control structures, the Shell supports the if...then statement, for and while loops, and the case statement. A limited form of exception handling is also available. For data structures, the Shell provides simple variables, parameters, and the Unix file system. The Shell provides a pattern matching facility for generating filenames. It also has a macro substitution facility.

The language provided by the Shell is rich enough for "control" programming, where the primary task is to invoke existing tools. It is not unusual for a program to be implemented initially by a Shell procedure (possibly augmented by a few specially written C programs) and then later rewritten in C. This allows easy experimentation, and as well as easy construction of "one shot" tools.

As an operating system interface, the Shell provides access to many of the features of Unix. The Shell allows the user to perform computations in the background. The Shell has a facility called pipes, which allow the output from one command to be used as the input of another command; a similar mechanism allows input and output to be redirected from or to files. This I/O flexibility is one of the powerful connective mechanisms that Unix uses to allow programs to be put together easily from component parts.

Source Code Control System

Many problems that arise during the software maintenance phase are due to a lack of adequate control and organization of program source code and documentation. The Source Code Control System (SCCS) is one of the best known systems for dealing with this problem [Glasser 78].

SCCS can be thought of as a database facility for textual material. It provides facilities for storing, changing, and retrieving all versions of a

Existing Advanced Programming Environments

text file. When changes are made, SCCS records the changes, but it does not discard the old text; this allows SCCS to retrieve any version of the text. It also asks users for an explanation every time a change is recorded, and saves this information along with the changes.

SCCS provides a tree-like structure for handling versions. In the simplest (and most common) case, the tree reduces to a straight line: each version is always an update of the previous version. Tree structures can arise for numerous reasons: a software system may evolve into several separate versions, or it might be necessary to provide temporary versions (e.g., temporary bug fixes), which may be replaced when a later "real" version is released.

SCCS does much work to ensure that nothing disastrous happens to the maintained text. It handles synchronization of multiple readers and writers, as well as attempting to protect users against interruptions or crashes. SCCS uses the operating system protection mechanism to control the creation and destruction of text. When an SCCS database is initialized, a database administrator is selected. The administrator is responsible for providing users with access privileges to the database.

SCCS has a facility for identifying the version of text file, even if the text file has been converted by some processor into a different form. For example, a source code file can be identified, as well as a load module and an executable file. On some systems, where all code is maintained with SCCS, users can easily identify the version of any program, without examining the source code.

Make

Make is a facility for rebuilding a software system automatically and incrementally. For example, if a programmer changes one module, invoking Make will rebuild the whole system, but it will only redo those parts that were affected by the change. Make is very simple (and relatively inexpensive) to use, yet it eliminates a large amount of routine, error-prone, and time consuming work for the programmer.

Make can also be used to perform a variety of other tasks associated with software systems. For example, a makefile can contain information which will test a system (if the testing can be done automatically) or perform other housekeeping tasks, such as deleting unnecessary files.

Existing Advanced Programming Environments

Other tools

As might be expected in an environment touting the toolkit approach, Unix provides numerous other tools. These tools can be divided into two classes: programmable and non-programmable. By a "programmable tool," we mean one that has some sort of input or command language (i.e., something beyond command-line options). We do not include bona fide programming languages here, for they are more than just tools.

Some of the popular programmable tools include:

<u>m4</u>	- a macro preprocessor
<u>yacc</u>	- a parser generator
<u>lex</u>	- a lexical analyzer generator
<u>awk</u>	- a pattern scanning and processing "mini" language
<u>grep</u>	- a generalized regular expression pattern matcher
<u>sed</u>	- a script-based, stream-oriented editor

Some of the popular non-programmable tools include:

<u>head</u>	- print first n lines in a file
<u>tail</u>	- print the last n lines in a file
<u>wc</u>	- count the number of characters, words, and lines
<u>diff</u>	- differential file comparator
<u>man</u>	- find and print information from the manual
<u>sort</u>	- sort or merge files
<u>find</u>	- find files with specified attributes

Existing Advanced Programming Environments

7.1.2 INTERLISP

Interlisp is a programming support environment based on the Lisp programming language [Teitelman 78], [Teitelman 81]. It is considered by many the most sophisticated programming support environment in existence.

Philosophy

The basic tenets of the Interlisp philosophy have been stated in [Teitelman 81]:

- The Interlisp environment is aimed at research and experimental programming, rather than production programming. The language and environment provide for a great deal of flexibility, allowing for growth necessary in experimental systems.
- Interlisp tries to do as much work for the programmer as possible. Its designers were willing to expend computer resources to improve human productivity.
- Interlisp is a system for experts. Its designers believed that users would prefer sophisticated tools, even at expense of simplicity. Hence, mastery of all of the tools and facilities is quite difficult, but achieving competency pays off in the long run.

Key characteristics

- Interlisp is a programming support environment built around one language (Lisp); much of Interlisp itself is coded in Interlisp.
- The Interlisp system is well integrated. All of it "lives in one environment" (i.e., one address space), and there is a great deal of sharing (hence dependence) between different parts of the system (e.g., tools share common data structures and subroutines).
- Interlisp is a flexible and extensible system. It contains a large number of parameters and "hooks" (in the tools and in the underlying interpreter) as well as sophisticated macro facilities. A good deal of flexibility is due to the inherent flexibility of the Lisp language itself.

Tools

This section discusses some of the more notable Interlisp tools.

Existing Advanced Programming Environments

The file package

The Interlisp programmer views Interlisp as a "residential" system -- all relevant programs and data structures are kept inside the Interlisp environment. All actions and changes are also relative to the current environment. This information usually must be saved, at some time or another, on a file (provided by the resident operating system). For example, functions used by an Interlisp programmer may be kept in several files. When the programmer changes a function, the new version will be present in the Interlisp environment, but will not be in the file. These changes must be made to the appropriate file or else they will disappear, since the residential environment "disappears" when the user exits from the system. The Interlisp file package provides a mechanism that keeps the residential environment consistent with the permanent file system. This mechanism can also be used directly by the programmer to "checkpoint" a running program, saving necessary information to allow recovery from a crash.

Masterscope

Masterscope is an interactive program for performing program flow analysis and cross referencing. Masterscope works in an incremental fashion: it maintains a database of the state of the "world" (i.e., what the programmer is working on), and updates it when the programmer makes a Masterscope request. Masterscope is well integrated into the Interlisp environment; not only can it provide information, it can be used to apply "program transformers" based on the results of a request. Masterscope goes a step beyond other cross reference tools by allowing a request to include a transformation based on retrieved information. For example, the request

rename the function oldcommand to be newcommand

renames the function "oldcommand" to be "newcommand," and then changes all function references to reflect the new name. Note that other references to the symbol "oldcommand" are not affected.

Masterscope uses a number of heuristics and assumptions to circumvent complexity problems. It also knows about the assumptions it is making, and can warn programmers about those assumptions.

DWIM (Do What I Mean)

DWIM is a sophisticated spelling and syntax error corrector. It is a unique system; there appears to be no other programming support environment with a similar facility. DWIM can perform different types of transformations: Some it performs automatically; others, it first asks the user for permission. When an error is fixed, computation continues; but first, DWIM also repairs the cause

Existing Advanced Programming Environments

of the error (wherever possible).

DWIM can fix spelling errors. It bases its corrections on a spelling list of possible words and a "closeness" metric for comparing the misspelled word with words on the spelling list. DWIM has a number of rules for fixing other types of errors, including certain types of syntactic and semantic errors.

DWIM provides the programmer with a good deal of control over the correction process. DWIM can be turned off entirely. It can operate in "TRUSTING" mode, where DWIM can make corrections without programmer approval, or in "CAUTIOUS" mode, where approvals are necessary. By default, when DWIM asks for the programmer's approval, it waits a prespecified amount of time, and then makes the correction automatically if no response has been received.

The DWIM facility is also directly available to the programmer. For example, the programmer could incorporate spelling correction into input processing for an application program simply by providing a spelling list and setting desired parameters.

Advising

Advising is a simple and clever mechanism. It essentially allows modification of a function without knowing how the function works, or even what it does (i.e., it treats functions as black boxes). It does this by modifying the interface between functions (and not the function itself), allowing arbitrary actions to be performed when a function is entered or exited. While this technique is not recommended for writing permanent code, it is useful for debugging and also for trying to understand how a program works.

Programmer's Assistant

The programmer's assistant is an intermediary between the programmer and the Interlisp system. It records all of the programmer's inputs, a description of the side effects of each operation, and results of each operation. The programmer can refer back to a previous action (or group of actions). A previous action can be printed; it can be redone; it can be redone with modifications (e.g., with different parameters, with errors corrected); or it can be undone.

The facilities provided by the assistant are tremendously useful in helping the programmer. It goes without saying that the capability to "undo" actions is a simply fantastic feature, not available in any other programming support environment. The "redo" capability can be thought of as a simple learning mechanism, putting a good deal of intelligence in the hands of the programmer.

Existing Advanced Programming Environments

Like all of the other tools in the Interlisp environment, the programmer's assistant mechanism is available to Interlisp programs. This allows an applications programmer make the redo and undo features available to the user.

7.2 CURRENT RESEARCH PROJECTS

Many research projects were reviewed in the course of this study. Most were reviewed via the technical literature or attendance at conferences and workshops. A number of key project teams were visited in person for detailed discussions. Rather than summarize all of the many projects investigated, we limit the discussion here to three important research efforts that have influenced our ideas regarding software maintenance tools. In addition, we note ongoing work at AI&DS that has resulted in a preliminary design and some key techniques for an intelligent program editor [Shapiro 82].

7.2.1 Stanford Verification Project

Goals

The Stanford Verification Project is concerned with the design and implementation of languages and systems that support the development of reliable software. The goals of this work are tools that support application of verification techniques and related methods throughout the development of programs, from initial design to testing and maintenance, and the methodology of their use. The tools are to be integrated with other components of the programming environment. A further goal is the development of a methodology of specification and annotation of programs.

Approach

The development of a high-level specification language tailored to current high-level programming languages is central to this research effort. The annotation language Anna extends the programming language Ada by addition of formal comments. Formal comments are constructed with the same precision as the program text proper; they are intended to be processed by advanced tools including a verifier, debugger, compiler, and documentation system.

The tools under development are being integrated in a prototype programming/verification environment. The environment can be adapted to different programming languages, and is extensible so that new tools can be added as they are developed. Emphasis is given to the interface between the various tools and the user interface.

The tools run off a common internal representation of programs and specifications. Most of the tools depend on a capability for mechanical

Current Research Projects

reasoning. The theorem prover is being designed in such a manner that it can be used by other tools as a subsidiary tool as well as by the user directly; accordingly, it includes components that perform simple reasoning tasks totally automatically, and an interactive component for guidance in more difficult proofs.

The system designs and implementation currently under way are intended to be language independent; the primary target, however, is the programming language Ada. The system is intended to support various forms of formal program validation techniques, from test case generation to symbolic debugging, analysis for specific run-time errors and formal verification.

Current status

The design effort for the verification environment benefits from the experience gained from using the Pascal verifier previously developed at Stanford. A preliminary design of Anna has already been published [Krieg-Bruckner 80]; a revision of this design and a formal definition of the language are in preparation. Some parts of the environment are operational in experimental form (e.g., syntax-driven display editor, Ada compiler). Currently, the integration of parts of the compiler into the editor (static semantic analysis) is under investigation. Several special analyzers are under development, e.g. a deadlock checker for systems of Ada tasks. The theorem prover component of the Pascal verifier has been re-designed, and languages for expressing specification concepts and guiding mechanical proofs are being developed.

7.2.2 Programmer's Apprentice

Goals

The Programmer's Apprentice Project (PA) is a research effort at MIT looking into the area of program understanding [Rich 78]. The goal of the project is to support all phases of program development by creating a system which models the organizing principles of code. The research takes the viewpoint that programs are structured objects (at both low and high levels of resolution), whose components can be represented explicitly. Given this representation, a powerful capability to analyze and manipulate code has been developed. There are two additional emphases in the PA research: an incremental model of program development, in which the Apprentice assists and possibly advises the programmer at each step of the way, and an analysis of existing systems into their component parts (in order to bring other programs into the domain of the Apprentice).

Current Research Projects

Approach

The Apprentice views the primitive building blocks of code to be stereotyped patterns of data and control flow. It uses these to represent slightly more complex constructs such as iterations and expressions. The Apprentice distinguishes several types of patterns in this domain, including: generators; filters and terminators for describing looping behavior; and compositions, predicates and conditionals for expressing straight line code. (Generators produce a sequence of values, filters restrict them, compositions linearize or segregate operations by data flow.) Using these concepts, programs can be parsed into a collection of simple purposeful parts.

At the next level up, the Apprentice considers programs to be largely composed of cliches, meaning the typical programming actions that form a major part of every programmer's repertoire. Examples of cliches include splice-in operations, list insertions, and hash tables abstractions, as well as a queue and process strategy, which is often used as to drive the function of many routines. Cliches are built out of the data and control flow primitives described above. One cliche may include another, or two may overlap, but the important point is that cliches organize large chunks of code into well understood entities.

The representation which captures this analysis is called a plan. Plans are an alternate method for expressing programs; they provide a representation that does not obscure program features critical to analysis. Plans have several important attributes:

- They ignore the way in which control and data flow are implemented. It makes no difference if a program uses conditionals or gotos, it still maps into the same plan. Similarly, all possible methods of using variables to propagate values are judged equivalent.
- They associate related segments of code which may have been separated in the original text. The fact that one piece of code outputs data which another consumes is a evidence that both are working toward some unified purpose.
- They are partitioned into fragments which have stereotyped behaviors. This means that plans embody a simple semantics for the low level structures found in program code.

Applications

The Programmer's Apprentice uses its knowledge of plans to support a number of operations on code. For example, during the editing process, the Apprentice can perform a semantic consistency analysis every time the programmer introduces a change. This allows it to recognize a permutation of primitive

Current Research Projects

operators (e.g., a car for a cdr in Lisp) as an error in the implementation of a specific aspect of a hash table. The Apprentice can also carry out some manipulations automatically. For example, it will be able to rewrite a sorted file abstraction to operate on keyed data. This transformation can be accomplished in part by filling in vacant slots in cliches.

Other applications have significantly used the recognition capabilities of the Apprentice. A system for understanding bugs was developed which first recognized the type of function that was being performed, and then analyzed the error [Shapiro 81]. A system which will produce high quality assembly code, using a plan based analysis of the task, is under development. There have been additional efforts within the Programmer's Apprentice project in automatic documentation and semi-automatic translation which rely heavily on plans.

Current status

At the current time, the Programmer's Apprentice project is primarily concerned with developing a large scale, integrated system that demonstrates its ideas. The first research task along this line is to develop a recognition engine which can break programs down into collections of cliches. This problem has been cast in the format of a sophisticated pattern match of a plan against a library of known cliches. The system which produces those plans given the text of programs exists in implemented form, and an extensive library of cliches has recently been published [Rich 81].

In terms of applicable techniques, the full impact of the Programmer's Apprentice technology is a number of years away. This is true in part because the Apprentice is a computationally complex system, but more importantly because it requires a tremendous knowledge base before it gains a reasonable coverage of programming activities. The Apprentice project also has a number of research issues to pursue in order to adequately support incremental program development.

In terms of the techniques which might be currently applicable in Ada environments, the analysis of programs into plans is a candidate. A knowledge based editor, which aids in the construction of programs from cliches, has been implemented [Waters 82]. A template based editor could be written to build cliches (or other structures) using the vocabulary of plans. However, it might be more appropriate to think in terms of programming support tools which use the notion of cliches, but forgo the generality inherent in the Programmer's Apprentice by using less sophisticated representations.

7.2.3 Gandalf

Current Research Projects

Goals

The Gandalf project at Carnegie-Mellon University is aimed towards an integrated programming support environment to improve the efficiency and reliability of the software development process [Habermann 82]. Gandalf goes further than trying to help the individual programmer working with a single program: It also addresses the issues of project management and version control. However, instead of building one programming support environment that can handle these problems, the Gandalf project is trying to build a system that can generate support environments for different programming languages. A Gandalf environment (i.e., an environment generated by the Gandalf system) provides a uniform interface for the programmer which is based on the syntax of the target programming language and other tools that are part of the Gandalf environment.

Approach

To generate a Gandalf environment, the Gandalf system requires a formal description of the abstract and concrete syntax of the target programming language, as well as any necessary semantic routines. Using this formal description, a system called ALOE (A Language Oriented Editor) is generated [Medina-Mora 82]. (Note that the generation process is concerned primarily with the incremental programming component of Gandalf, and secondarily with the version control component; however, the project management component is independent of the target language, and so it does not have to be regenerated for each Gandalf environment.) ALOE is a syntax oriented editor which has commands for constructing and editing a Gandalf database. All components of a Gandalf environment (incremental program construction, system version control, and project management) operate on a common database. Thus, only one interface is necessary for all steps in the programming process.

As a syntax oriented program editor, ALOE allows the programmer to talk about program in terms of syntactic constructs, rather than lexical constructs. This prevents a programmer from entering a syntactically incorrect program. ALOE provides incremental program compilation; programs are compiled as they are created or changed (without invocation of a compiler by the programmer). Thus, compile time errors are caught almost immediately. Programs can also be run and debugged from ALOE. Initiating the execution of a program takes little time, since the program will usually be fully compiled, due to incremental compilation. Breakpoints and debug statements can be inserted directly into the text of the program, thus providing a simple yet effective debugging mechanism.

ALOE can act as a syntax editor to interface to the System Version Control Language (SVCE) [Kaiser 82]. This language provides the programmer with a way of specifying the relationships between program components and versions. The version control specification techniques grew out of work presented in [Tichy 80].

Current Research Projects

Finally, ALOE provides an interface to the Software Development Control (SDC) component of Gandalf, which provides project management facilities. SDC handles a number of tasks, including synchronization of multiple programmers working on one project, change logging, and protection and authorization issues. The SDC component controls the state of the software; by using the version control facilities, it can ensure that software is in a consistent state before the software system is rebuilt.

Because of the integrated approach taken by Gandalf, the programmer may not even be aware of the version control and project management components. ALOE is capable of automatically invoking these facilities at certain times (e.g., automatically recompiling a module that is dependent on a module that has just been changed).

Current status

A prototype Gandalf environment for a dialect of the C programming language currently exists and is being experimented with. A Gandalf environment for the Ada language is anticipated; however, the complexity of Ada may complicate the generation of such an environment. Also, some of the techniques developed for the Gandalf prototype may need to be discarded for an Ada Gandalf. For example, some of the version control mechanisms could be replaced by constructs provided directly by Ada; the central database provided by an APSE may be used instead of the current Gandalf database. But overall, the Gandalf paradigm seems quite powerful, and many of the techniques developed in the Gandalf effort will most likely make their way into future versions of APSEs.

8. DESIGNING ADVANCED TOOLS TO EASE MAINTENANCE

One of the major goals of this study was to develop designs for maintenance tools aimed at improving productivity and reliability. This chapter presents some general issues that are important considerations in designing tools. (Designs for specific tools are presented in the following chapter.) While there are many issues that might be discussed here, this chapter focuses on two particularly interesting topics. The first is software productivity, which is a topic of interest in most studies of the software life cycle. The second topic covers several trends and techniques that may have an important impact on advanced tools.

8.1 SOFTWARE PRODUCTIVITY

Software productivity refers to the rate at which software is developed and maintained. Traditionally, productivity has been measured in "lines of code per man-month" [Gloss-Soler 79]. A more qualitative definition defines software productivity as the rate at which useful work is performed on software. Because the development and maintenance of software is a labor-intensive process, software tools aimed at improving the process should take into consideration factors that might directly influence productivity.

[Reifer 82] has identified several of these factors affecting software productivity. He calls these the "five Ms" :

1. management - Administration and organization are necessary in any programming environment. Management functions are generally performed by people. However, some of these functions can be performed by (or with the aid of) software tools (e.g., keeping people informed of schedules, keeping programmers up to date on available tools and subroutine packages, making sure that software is checked for adherence to standards).
2. motivation - Simply giving a tool to a programmer is not sufficient. Management must motivate its use, by explaining why the tool should be used, and providing positive feedback when it is used. The tool itself should give some feedback to the user. For some tools, such as a display editor, this requirement is obvious. For other tools, such as a program checker, feedback is often omitted under certain conditions (e.g., when the tool encounters no errors).
3. methodology - Use of methodologies (in all phases of the software life cycle) will have a positive impact on the process, by providing a systematic way for performing sets of tasks. For example, various programming methodologies (such as functional decomposition, data flow design methodology, and data structure design methodology [Bergland 81])

Software Productivity

provide techniques and rules for designing and coding software. Promoters of specific methodologies sometimes promise miracles for faithful followers. However, any methodology (within reason) is generally better than no methodology at all, since the simple step of systematizing the process is often sufficient to increase productivity. Acceptance of a methodology (by both management and technical staff) is generally the most difficult step; once in place, it is easier to replace an ill-suited methodology with a more appropriate one.

The design and selection of methodologies is still very much an art, and must be performed by people. Once a specific methodology (or set of methodologies) is selected, software tools should be provided to aid each step of the methodology.

4. mechanization - The primary function of software tools is to mechanize (i.e., automate) the software process. Just as methodology helps systematize high level tasks (like designing the control structure of a program), mechanization systematizes lower level tasks that are best performed automatically (e.g., checking standards, logging changes).
5. measurement - Measurements of performance are necessary to judge both programmers and software. Used appropriately, this data can be used to objectively improve the software process. For example, performance statistics for a programmer can be useful in directing an educational program; statistics about the quality of a program can be used to help decide if the program should be modified or rewritten.

Few individual software tools address all of these issues. However, designers of programming tool sets should take all of these issues into account. While the focus here was on the "five Ms" with respect to software tools, the "Ms" can also be considered from the viewpoint of managers or programmers; this would be an appropriate topic for a project studying the "human component" of the software life cycle.

8.2 TRENDS FOR ADVANCED TOOLS

Turning from the issue of productivity, this section addresses some slightly more technical issues. In surveying existing tools, as well as research efforts, some particularly important trends and techniques have surfaced, namely: integration, intelligent user interfaces, domain knowledge and reasoning, history, and incrementalism. These trends represent important paradigms for the entire programming process, capable of forming the basis of a new generation of programming tools. Other than that, these trends are fairly dissimilar, varying in scope from the very broad to the fairly specific. The remainder of this chapter discusses these trends in more detail. The following chapter presents designs for a number of advanced tools for the maintenance environment which make use of all these concepts.

Trends for Advanced Tools

8.2.1 Integration - Tools That Work Together

There is a saying that "...the whole is more than the sum of its parts." This notion of synergy is important in the design of software tools. When several tools work together, they may provide something that neither one could alone provide. The term integration is used here to refer to the degree of synergy and close coupling between tools. Tools in a well integrated system exhibit a large degree of synergy (as a result of working well together). Since synergy results from interdependencies, integrated tools are likely to share information, share common procedures, or provide complementary functions. Systems such as Interlisp [Teitelman 81] and Unix [Kernighan 81] owe a large amount of their success to their integration.

Well integrated systems provide several important advantages:

1. Human comprehension is aided by the uniformity provided by a well integrated system. A consistent underlying philosophy aids users in making inferences about how the system works.
2. An integrated tool set allows one to put tools together quickly in order to perform tasks that may have not even been envisioned by the system designers. This benefit is well known by Unix users.
3. Integrated tools work together, allowing more efficient and effective performance. Efficiency is gained when tools can make use of each others' work, eliminating redundant computations; for example, symbol tables created by the compiler can be used by debuggers, linkers, cross-reference listers, etc. Effectiveness is increased when tools can make use of each others' information; for example, a compiler may be able to apply optimizing code generation strategies by getting information from the program verifier that the compiler cannot deduce by purely syntactic means.

Related to integration is the idea of completeness. Completeness means that the user should be able to do everything that might be needed. The beauty of an integrated system is marred when a user has to expend a large amount of energy to do something that is conceptually easy but the system does not allow. An integrated system should provide a mechanism to allow the user to perform these sorts of tasks. For example, the vi text editor [Joy 80] is a display oriented editor that provides a basic set of text editing functions. It provides an escape mechanism that allows the edited text to be fed to other programs (which are independent of the editor); e.g., text formatters. This is easier (and faster) for the user than saving the edited text, exiting from the editor, running the program, and then restarting the editor with the modified text. As another example, the Interlisp system allows certain common monitor-level commands to be performed without leaving the system; to perform other commands, there is a simple interface that creates a new process running the operating system's command processor, allowing the user to execute

Trends for Advanced Tools

arbitrary commands and then return to Interlisp without any loss of continuity.

Designing a consistent yet usable system requires a great deal of ingenuity and insight on the part of the designers. But the effort does pay off; look at the popularity of Unix. The APSE (Stoneman) also specifies an integrated tool set; the success of this requirement will be seen as APSEs are implemented and used.

8.2.2 Intelligent User Interfaces

An intelligent user interface can be seen as a programming support system that provides (in varying degrees) some of the intelligence and assistance that a human programming assistant might provide. An intelligent user interface not only makes life easier for the programmer; it helps increase programmer productivity and software reliability.

What kinds of features might an intelligent user interface provide?

- programmability: The user interface in a programming support environment should provide the programmer with tools for programming his own work. Many programming tasks are repetitious or systematic. The programmer should be able to write a simple program to perform a task (unencumbered by the burdens of modern programming languages), or, better yet, have the system learn how to perform the task by watching the user perform it (e.g., [Waterman 78]). Also, an initial set of high level commands, performing common sequences of lower level commands, should be provided (e.g., a command to compile, link, and execute a program).
- error prevention: By making "bad" things hard to do, it is less likely that they will be done inadvertently. [Conversely, good things should be easy to do.] Warnings about dangerous actions, before they are performed, further reduce the chances of error.
- error detection and correction: It is not too difficult to catch many types of errors automatically. Every attempt should be made to catch errors as early as possible; the later an error is detected, the more expensive it is to fix. Error diagnostics should be meaningful to the user, not only to the person who wrote them. Some errors, especially silly, careless ones (e.g., spelling errors), can be corrected without too much difficulty.
- recoverability: If an error is made, the user should be able to recover as easily as possible. The system should have safeguards to provide the user with certain paths of recourse, e.g., by allowing actions (such as deleting a file) to be undone. "Forgiveness" is important; making a blunder is bad enough; one should not have to spend hours or days to right it.

Trends for Advanced Tools

- active help: If the user repeatedly does things incorrectly, there may be no need to wait until help is requested. In many cases, the user may not be aware that help exists, or may not know how to ask for it. Help should be offered automatically.
- non-interactive mode: The system should be able to function without human intervention if necessary. If a programmer leaves the terminal while performing a task, there is oftentimes no need to bring things to a halt when only trivial human input is needed.

8.2.3 Domain Knowledge and Reasoning

One way of making tools and systems easier to use is to endow them with a certain degree of "intelligence." Most tools do not have any idea what they are doing; for example, text editors don't know anything about editing computer programs, nor do compilers understand what programs are supposed to do.

Domain knowledge and reasoning can be quite useful in a programming support environment. This is nicely illustrated with an example from an analogous situation. Suppose you had a technical manuscript that needed to be typed. If you gave the manuscript to a typist who spoke no English, you would expect, at best, a word-for-word typewritten copy of the manuscript. If you gave it to an English-speaking typist, you would hope that simple errors, such as misspellings and punctuation errors, would be fixed. If you gave it to an English teacher moonlighting as a typist, you wouldn't be surprised to find that some of your prose had been changed and improved upon. And if you were lucky enough to find a typist familiar with the domain of discourse (of the manuscript), you shouldn't be surprised to find factual errors corrected.

The problem of getting the manuscript typed with the best possible result is similar to the problem of writing a program. You select some type of editor to use for entering the program text. A standard text editor would be comparable to the non-English-speaking typist: Text is entered exactly as typed, with no enhancements. The English-speaking typist could be compared to a syntax-oriented editor: one which can eliminate syntactic program errors and misspelled keywords. The remaining two typists have a fair degree of knowledge, and understand how to apply it. The English teacher/typist knows about the language itself (rather than content of what is being said). This situation is comparable to a programming language-specific editor, which applies knowledge about the domain of programming; the editor can help with general programming techniques, can catch certain types of semantic errors, can make style suggestions, and can improve the general flow of the program. The technical typist who understands the content of what is being said is analogous to an editor which utilizes knowledge about the application domain. It can help with domain specific techniques, such as algorithm development, and can catch certain kinds of pragmatic errors which are dependent upon the specific application domain.

Trends for Advanced Tools

So, in a programming support environment, it is desirable to have two types of expertise, programming expertise and application expertise. An "ultimate" goal might be to endow the system with expertise equalling that of a human; the system would exhibit programming expertise comparable to that of a computer scientist, and application expertise similar to that of a domain specialist. Note that in a programming support environment, the latter type of knowledge is more specialized, hence less widely applicable (a new knowledge base is needed for each application area).

To construct these types of capabilities, both domains have a need for representation and modelling of human users. It is necessary to understand the programmer's actions (what he is doing) and intentions (what he will be or wants to be doing). Knowledge cannot be applied effectively unless there is some technique for deducing what is happening. For example, an editor incorporating programming domain knowledge needs to know what parts of a program the programmer will be writing or changing, as well as the (expected) effect this will have on other parts of the system. An editor incorporating application domain knowledge needs to know what techniques the programmer will be utilizing, as well as the type of output and results expected.

The use of domain knowledge and reasoning in the programming environment will drastically change the whole concept of programming. It will allow the software tools to truly help the programmer, freeing the programmer to concentrate on higher level issues.

8.2.4 History - Keeping Track of What Has Been Done

The notion of computational history refers to the information available during the course of some computation. For example, when using a text editor, the history includes the editing commands as well as the inserted and/or deleted text; when using a compiler, the history includes the original source code, the parse trees, parse tree transformations, and generated code. Some of this information has no long term value beyond immediate consumption by a program; but much of the information is quite valuable, either because it is expensive to recompute (e.g., parse trees for a large module) or because it cannot be recomputed (e.g., record of all operations performed by the user). The predominating practice of discarding this information must be re-examined. It is becoming apparent that effective programming environments must capture and provide access to this type of information.

There are numerous reasons why history is a necessary ingredient for advanced programming environments. First of all, sophisticated programming environments must allow programmers to make changes incrementally, so that the cost of making small changes is small (see next section for more detail). To accomplish this, intermediate results of various system tools and utilities (e.g., compilers, linkers) must be kept around. Another need is accountability: records of all important activities should be maintained so it

Trends for Advanced Tools

can always be determined what has been done and who has done it. Important activities include things like changes to code, document updates, system builds, etc. From a user interface perspective, preservation of a history is also desirable. Some programming systems, such as Interlisp, allow the user to see a record of what has been done, and allow transactions to be "replayed." Finally, history is necessary for the application of programming domain knowledge and reasoning (see the previous section). To understand what the programmer is doing, it is necessary to understand the context in which the programmer has been working.

The Interlisp programming environment has a history mechanism which is well integrated the programming environment, and APSEs have databases that also provide the requisite mechanisms.

8.2.5 Incrementalism

Support of incremental change is vital for the maintenance of all but the smallest systems. It is unacceptable and unnecessary to require a whole system to be rebuilt each time a small change is made: unacceptable because the cost is too high; unnecessary because changes usually leave many parts of the system unaffected.

The move toward building systems that handle incremental change has been slow, primarily since it is (in general) more difficult to build tools that are incremental. There are several problems. First of all, new algorithms may have to be devised, or old algorithms modified, in order to handle "batch" requests as well as incremental requests. Another problem is lack of information: most tools throw out information as soon as they are done with it, rather than leaving it around for future reference. An example of this is symbol table information, which the compiler builds up for each module and then usually discards. This means that the symbol table must be rebuilt for each recompilation, even if code changes had no effect on it.

Incrementalism is a technique vital for the development and maintenance of large systems, yet few existing programming tools make use of it. Aside from some research on incremental techniques for syntactic parsing, most attempts at incorporating incrementalism have been somewhat ad hoc (and less than generally applicable). The idea of incrementalism falls out naturally when some of the other techniques discussed earlier in this section (e.g., history) are incorporated into the programming environment.

9. ADVANCED TOOLS FOR SOFTWARE MAINTENANCE

This chapter describes several tools that could be particularly important with respect to the future development of maintenance programming environments. These tools are not meant to be an exhaustive list of useful tools; such a list is beyond the scope of this project. Nor are these tools meant to represent an integrated or complete set: it is assumed that they will reside in a programming support environment such as the APSE, which already provides the basic necessities. Each of the selected tools satisfies the following criteria:

- the tool should have potential for making a significant impact on the maintenance process;
- the tool should be either unavailable (i.e., no one has ever built one) or else not widely known and used;
- the tool should benefit via application of advanced technologies (e.g., AI)

There are other tools satisfying these criteria which have not been listed here. The goal here was not to catalog all possible tools, but rather to select a particularly important subset.

As these tools are described, it may appear that their goals are often ambitious. However, each tool has components that are immediately realizable, as well as components that are research efforts. Thus, each tool might be considered a family of tools, of which some members are feasible in the short term, while others members will take more time.

This chapter contains nine sections, each of which describes one tool. Two of the tools, the Programming Manager and Intelligent Editor, may be classified as comprehensive or meta-tools in that much of their function is to provide a standardized interface to other tools (including some of the other tools discussed here). Four other tools are much more focused on one problem or aspect of software maintenance; these tools are the Metrics Tool Set, Style Analyzer, Change Propagation Detector, and Test Case Analyzer. The breadth and nature of the activities of the remaining tools are somewhere in between the comprehensive and focused tools; these are the Documentation Assistant, Intelligent Tutor, and Annotation Language.

The proposed tools are given names that attempt to describe their functions; there are no clever acronyms (but suggestions are welcome!). To differentiate between generic tools and specific instances of the proposed tools, the

convention of treating the names of specific tool instances as proper names is used (i.e., first letters are capitalized). Each tool description contains the following parts:

<u>abstract</u>	- a short overview
<u>need</u>	- the problem the tool is aimed at
<u>approach</u>	- how the tool tackles the problem
<u>use</u>	- how the tool will be used
<u>examples</u>	- more specific instances of tool use [optional]
<u>issues</u>	- miscellaneous issues relating to feasibility, etc.
<u>time line</u>	- representative examples of what could be developed in the following time periods:

short term (under 3 years)
medium term (3 - 7 years)
long term (over 7 years)

Programming Manager

9.1 PROGRAMMING MANAGER

abstract

The Programming Manager is a tool that will help improve the program development and maintenance process by ensuring the systematic application of managerial and technical policies and methodologies. It will also provide a basis for transferring procedural and heuristic knowledge between programmers.

need

There are three problems of particular importance in the maintenance programming environment:

1. In any programming organization, there exist management policies and standards that programmers are supposed to follow. Organization-wide programming standards help promote quality and reliability, and ensure that programmers can be moved between projects without significant retraining. Unfortunately, the complexity and volume of such policies and standards can be overwhelming, and so are often ignored.
2. When working with large software systems, programmers spend a lot of time learning (but then forget) how things work. During the process of modifying and debugging code, a programmer really begins to understand the minute details of the code. This knowledge is lost as soon as the programmer stops working on that section of code or leaves the organization. There is a real need to capture at least some of this information. The usual routes for recording information - manuals, memos, reports, etc. - are often not considered appropriate repositories for this information, which may be low level, obscure, and often heuristic.
3. In modern programming environments, programmers are faced with a large number of tools, each applicable to certain parts of the software life cycle. Some of these tools are absolutely necessary, and each programmer quickly learns how to use them (e.g., editors and compilers), though many of their intricate commands or parameters may be soon forgotten. Other tools may not be used as often, and programmers may forget how to use them (e.g., macro preprocessors, debuggers). Some tools may provide functions that are not vital to the process of getting a system to work; hence, they are forgotten or ignored (e.g., style analyzers, execution time profilers). Programmers need assistance in making effective use of these tools. Manuals and on-line documentation are not adequate, for they provide information only when explicitly consulted and thus don't help in selecting tools that the programmer doesn't know about or has forgotten.

Programming Manager

approach

The Programming Manager is an expert system whose domain of expertise is the programming process. It will collect and maintain rules and information relevant to the process. Rules can range from administrative rules (e.g. when to file change reports) to technical (e.g. how to debug code); they can be used to structure the programming process, or they can be used to aid a programmer working on a complex module; they can be entered when the Manager is first deployed, or they can be entered on the fly by a programmer as he learns how a module works. Information maintained will include records of what the programmer has been doing as well as the status of the target software. The Manager will use this knowledge to help the programmer through the morass of tools and policies typically found in a programming environment, thus permitting the programmer to work effectively in an environment that is too complex for people to thoroughly comprehend. Thus, the Manager is an intelligent user interface to a programming environment.

use

The Programming Manager's job is to both help the programmer and enforce policies. The Manager could function in several different modes: it could aid the programmer only when asked for help, in the meantime watching (and recording) all that the programmer does; it could aid the programmer when it deems necessary, e.g., when the programmer has neglected an important step; or it could lead the programmer through every step of the process (an excellent technique for training new programmers). One goal of the Manager is to be helpful without being obtrusive; the only time it will get in the programmer's way is when the programmer tries to do something that is not allowed (e.g., like releasing an untested version of a module) or fails to do something that is required. The Manager can work with the programmer during all phases of the software life cycle.

Rules are the basis of the Manager's power and flexibility. The rules can be organized in a hierarchical rulebase, thereby allowing the scoping of rules, based on the structure of the development or maintenance organization (e.g., there may be some government-wide standards, then DoD standards, then Air Force standards, then maintenance organization standards, etc.). Rule acquisition is a flexible process; rules can be placed in the rule base from the start, or can be acquired on the fly (during the programming process). Policy rules may be placed in the initial rulebase, whereas heuristic rules, which the programmer knows or discovers, are most naturally added during the maintenance process.

examples

The following examples show how the Programming Manager handles some of the problems addressed in the preceding need section. The examples are presented in terms of scenarios that a maintenance programmer might be likely to encounter. For the purposes of exposition, the dialog presented here is in

Programming Manager

natural language. In an actual implementation, the communication would be in a more restricted form.

Scenario: The programmer wants to make a change to a module.

Programmer: Edit <module>.

Manager: You must create a Modification Design Statement before changing <module>. Do you want to see the correct format?

Explanation: The Manager has found a rule which says that no modules can be changed until a design for the change has been created.

Scenario: The programmer has changed a module and wants the changes to become "official".

Programmer: Add <module> to program library.

Manager: You must first run the code auditor on <module>. Shall I do it for you now?

Explanation: The Manager has a rule that says all modules must be checked out by the code auditor before becoming part of the official program library. Since the Manager keeps track of the programmer's actions, it knows that changes have been made to the module and that the code auditor was not invoked. The Manager is capable of automatically invoking the code auditor if the programmer desires.

Scenario: Over a period of time, the programmer notices that changes to module X often cause problems to crop up in module Y.

Programmer: Module Y appears sensitive to changes in Module X.

Manager: OK, I will make a note of that.

Explanation: The Manager recognizes that the programmer is offering information, and so it records the knowledge for future use.

Programming Manager

Scenario: The programmer makes some changes to module X, and notices that strange results are being produced by module Z during testing.

Programmer: Module Z appears to be producing strange results.
What could be wrong?

Manager: Check module Y for errors introduced by the changes you made in module X.

Explanation: While analyzing the situation, the Manager noticed that module Y calls module Z. It also found the programmer's information regarding the sensitivity of Y to changes in X. Since changes were made to X, the Manager concludes that module Y is a good place to start looking for problems (the Manager will have also analyzed other modules; in this case, we assume that none of them was a likely candidate).

Scenario: It has been decided that maintenance programmers in a particular group must use a newly developed tool for analyzing program style.

Supervisor: Add this rule to the database: "Before a module is tested, it should be able to pass through the new program style analyzer without any complaints."

Manager: OK, I'll add that to the rule database.

Explanation: The Manager maintains information about how and when to use available tools. The addition of this rule will insure that programmers use the new tool (even if they haven't been previously notified of its existence).

Scenario: The programmer is ready to test a module.

Programmer: Test <module>.

Manager: You should run <module> through the new style analyzer before you do any testing.

Explanation: Because of the new rule added by the supervisor, the Manager automatically notifies the user that the new tool should be invoked at this point. As in previous examples, the Manager could have invoked the tool automatically, if desired.

Programming Manager

issues

Some of the technology required by the Programming Manager has been developed in the field of artificial intelligence; however, the necessary techniques have not been put together in one system, and careful attention must be paid to integration of techniques as well as human engineering aspects (which are particularly crucial to a system of this type). For the Programming Manager to be successful, it is necessary that it both perform its tasks well and interface with users in a natural fashion. Some of the technology required is still considered a research area in software engineering (e.g., models of how people program, programming methodologies, specification techniques).

The Programming Manager is a "meta" tool; its primary purpose is to (help the programmer) control the application of other tools. Research will be necessary to determine the meta-knowledge that must go into the system. Determining the required knowledge and meta-knowledge and selecting appropriate representation formalisms represent a large part of the research necessary to build the system. And as with any expert system, the acquisition of prerequisite knowledge will require a reasonable effort.

The Programming Manager can function as the basis for a distributed programming environment, where the programmer's immediate environment is separate from the development/maintenance environment. This provides flexibility, and also minimizes competition for computational resources between the Manager and the programmer's application.

time line

under 3 years

- script-driven (non-heuristic) control: The Programming Manager would initially have a simple control mechanism based on prestored scripts which would contain information on typical programming actions likely to be invoked by a programmer.
- history recording: All actions taken by the user would be recorded, for future reference by the user, for documentation, or for later use by the Manager.
- integration of many tools: Using the script-based approach, many small tools (such as those found in the Unix environment) could be integrated into the environment provided by the Manager.

Programming Manager

3 - 7 years

- rule driven: The Manager would employ a more flexible control technique, based on rules, that would allow it to adapt more readily to the environment, as well as provide the ability to handle sequences of actions taken by the programmer that were not anticipated by the authors of the rules.
- methodology driven: The programming methodology selected for the environment would be incorporated into the knowledge base of the Manager.
- models of tools: The Programming Manager would have formal descriptions of what each tool does, allowing it to select among tools more intelligently (i.e., not blindly applying rules).
- intelligent user interface: By "watching over the user's shoulder" to see what is being done, the Manager does not need the user to explicitly state what has been happening; it figures things out by inferring from the communication between the user and the system.
- simple inferencing: There will be a capability for making logical inferences based on rules, models of programming tools, a simple model of the programming process, etc.
- simple explanation: The Manager will be capable of telling the user why a particular choice was made.
- semi-automatic rule acquisition: When the Manager sees the programmer perform new steps that seem to have some importance, it asks the programmer the meaning of these steps, and can then add them to its knowledge base if requested by the programmer.

over 7 years

- methodology specification language: With a technique for the formal specification of methodologies, the Manager will be able to work with a definition of the methodology, allowing more flexibility and intelligence than the rule-based approach (which describes only the procedural aspects of the methodology).
- modelling of user actions: The Manager would use some model (or set of models) to understand how users might function in the context of a programming environment; models might range from relatively simple (model of how a person reacts to a stereotypical programming situation) to complex (model of human problem solving).
- programming domain modelling: The Manager would utilize models of the programming process, e.g., a model of debugging, model of program

Programming Manager

development by by stepwise refinement.

Intelligent Editor

9.2 INTELLIGENT EDITOR

abstract

The Intelligent Program Editor will provide facilities for manipulating programs at a number of conceptual levels, and act as a general interface to be used in conjunction with other tools in the programming environment.

need

Current programming environments use general purpose text editors for the manipulation of program text. Much is lost by not taking advantage of the structuring information inherent in programming languages. First, this information could be used to prevent certain types of errors (e.g., syntactic errors). Second, the information could be used to allow program manipulation based on language syntax and semantics (e.g., to delete a block of code, the user could say "delete this block of code" instead of saying "delete these N lines of code"). Third, the structuring information could allow the editor to serve as an interface to other language-oriented tools (e.g., debuggers). Finally, the information could be used to support incremental program development; the editor could tell the compiler which syntactic units have been changed, instead of forcing the entire module to be recompiled.

approach

The Intelligent Editor will provide an interface to program text, based on the syntax and semantics of the programming language. The Editor will aid the programmer by providing incremental checking for semantic completeness and consistency (thereby allowing errors to be caught early in the process) and by allowing the specification and manipulation of program parts by the use of program specific techniques. The Editor might also provide an interface to other language dependent tools proposed later in this chapter, e.g., the Style Analyzer (section 9.4), the Documentation Assistant (section 9.3), the Annotation Language (section 9.6), and the Change Propagation Detector (section 9.7).

The basis of the Intelligent Editor is the program reference language and the underlying knowledge representations of programs and documentation [Shapiro 82]. The program reference language provides a formalism for talking about programs; it was originally defined in [McCune 79]. It provides numerous methods for referencing programs: lexical, syntactic, contextual, historical, semantic, and pragmatic methods. The Editor extends work done on syntax directed editors and transformation systems. It makes use of techniques from software engineering and artificial intelligence.

Intelligent Editor

use

The Editor actually will provide a working environment for the programmer. Since the Editor and all the tools to which it allows access are based on the target language (Ada, in this case), the user is both protected and aided in the environment; it becomes easier to do things right (and harder to do them wrong). Since the Editor may act as an interface to other language dependent tools, as well as providing for direct manipulation of program objects, it will aid in many phases of the software life cycle. It will fit into any programming environment, but an environment such as an APSE allows for the full integration that makes the Editor so useful.

examples

The following examples demonstrate the capabilities of the program reference methods, as discussed in the preceding approach section. For purposes of exposition, the references are in natural language; in a production system, the references would be in a more restricted language.

lexical:	"the line containing the string 'for' "
syntactic:	"the <u>for</u> loop using the variable 'sum' "
contextual:	"the third <u>for</u> loop down from the current location"
historical:	"the last loop that was referenced"
semantic:	"the loop that accumulates the variable 'sum' "
pragmatic:	"the loop that computes the sum of the test scores"

issues

The design of the program reference language is a topic that must be further researched. The required integration between the Editor and the rest of the environment will pose some engineering problems, since each tool in the environment has its own model of the world. There is a fair degree of risk in building the Editor, but the potential impact is also significant. The idea of language oriented editors has been around for a while (e.g., in the Interlisp environment [Teitelman 81]), but there has been a recent flurry of work in design of syntax oriented editors (e.g., Mentor [Donzeau-Gouge 79]) as well as work in integrated environments (e.g., Gandalf [Habermann 81] and the Cornell Program Synthesizer [Teitelbaum 80]). However, these systems do not go very far in using knowledge about the semantics of programs and programming languages.

Intelligent Editor

time line

under 3 years

- syntax editing: The Editor would perform manipulations based on a formal syntactic description of the target programming language.
- simple interface to other tools: It would have the ability to invoke tools on all or part of a program, plus a limited ability to interpret results of certain tools (e.g., if the Style Analyzer complains, the Editor would keep track of the offending lines).

3 - 7 years

- simple semantic searching and editing: The Editor would utilize semantic information about the programming language to allow the programmer to communicate with the editor in terms of the language (e.g., "find the loop that accumulates the variable 'sum'").
- incremental checking and compilation: Programs would be checked for consistency and correctness while the editing is being performed; correct program parts could be compiled (incrementally) if they appeared completed.
- simple code transformation: A library of source-to-source transformations (e.g., changing a for loop to a while loop) is available upon request by the programmer.
- integrated interface to other tools: The Editor could automatically invoke tools and handle their responses; tool invocation could often be done without the programmer's knowledge.

over 7 years

- language-specific semantics for editing: The Editor would be capable of using a full set of formal language semantics for manipulating programs; this would be especially useful in the more complex parts of the language (e.g., tasking in Ada), where simple semantics would not be sufficient.
- domain-specific semantics for editing: The Editor would also employ knowledge about the application of the program, thus allowing manipulation based on what the program is doing.
- complex code transformations: A more complete set of transformations would be available, based on inferencing mechanisms, which would allow any

Intelligent Editor

transformation specifiable in terms of the language's syntax or semantics (e.g., change a set of procedures into a set of parallel tasks).

Intelligent Editor

The Programming Manager vs. The Intelligent Editor

Both the Programming Manager and the Intelligent Editor can serve as interfaces between the user and the programming environment, and in the tool descriptions that follow, one of these tools is often mentioned as a possible interface for other tools. This is not meant to imply that the other tool would not make an adequate interface; the purpose is to present an operational scenario, rather than enumerate all possibilities.

It is interesting to note the differences between the Programming Manager and the Intelligent Editor, when considered as interfaces. The Editor's domain of expertise is programs, and thus will provide a natural interface for other tools which may require just such expertise. The Programming Manager's domain of expertise is the entire programming process; but like any good manager, the Programming Manager should "stand back" when one of its "subordinates" has some particular competence. In an environment with an Intelligent Editor, there is no need for the Manager to replicate the knowledge about programs that the Editor has.

The Editor will provide a tightly coupled environment, where all the tools work together and share resources, as a result of their need for related types of knowledge. The Manager will handle a diverse set of tools, and will not require that there be any logical cohesion between the set of tools it manages. It can manage tools that manage other tools (like the Editor) in a variety of ways: by giving the subordinate manager full responsibility, by providing alternate access to tools managed by the subordinate, or by sharing knowledge and responsibility with the subordinate.

Documentation Assistant

9.3 DOCUMENTATION ASSISTANT

abstract

The Documentation Assistant will provide assistance to the programmer in creating, structuring, maintaining, and accessing software documentation, including requirements specifications, design documents, in-line comments, test documents, user documents, and maintenance documents.

need

Although documentation is one of the largest problem areas in software maintenance, there appear to be few generally available tools that support the tasks of collecting, maintaining, and utilizing documentation. In-line program comments have suffered at the hands of programming language designers, who have continued to neglect comments, treating them as ignorable text which happen to be randomly interspersed with the code. Other documentation also takes back seat to program code, and it is difficult to keep the documentation up to date with the actual code.

approach

The Documentation Assistant will provide a set of tools to help in the process of writing documentation, using documentation, and keeping documentation updated. These tools will provide support for incremental documentation (due to the incremental nature of maintenance, the only way to keep documentation "in sync" with the rest of the system is to update the documentation incrementally as other things change).

The Documentation Assistant will handle the two types of documentation found in the programming process: program comments and documents. The term "document" is used here to mean "non-program" text, including requirements, high-level design rationale, user manuals, test data, etc. Both types of documentation will be treated uniformly by the Documentation Assistant, which is somewhat different than the standard practice of treating program comments as "lower class" documentation. Each piece of documentation will be treated as a unique object. A program comment, previously considered to be a piece of text that happened to lie adjacent to some piece of code (in a source code file), will now be formally considered an object in its own right. All documentation will be attached to code, i.e., a link will be made from the documentation object to the code which the documentation references. Documentation objects are structured (e.g., via templates or frames). Each object will have a functional property associated with it which specifies the relationship of the documentation to the code (e.g., documentation may say what is done, why it is done, how it is done, or when it is done). The link will also have chronological information (so it can be determined if documentation was updated to reflect a code change) and author information.

Documentation Assistant

Tools for collecting and maintaining documentation could be invoked by the Intelligent Editor or by the Programming Manager. Use of the Documentation Assistant may require considerably more work on the part of the user than before, but this is not a problem inherent with the Assistant; rather, it is a function of the documentation process. Since programmers will be prompted incrementally for structured documentation, the process should be much less painful than current documentation methods. Tools for accessing documentation will provide users with better information than ever before. One accessing tool will be a simple, keyword-based, information retrieval system. Another tool will provide more sophisticated rule-based access of structured information (e.g., "retrieve documentation on why a change was made and who made it") [McCune 82]. The organization and structure provided by the Documentation Assistant will allow tools to make use of documentation in new and unique ways, e.g., using documentation links to track down implicit code dependencies. The Assistant is extendable, and could be used for record keeping in various phases of the software life cycle; for example, the Assistant could be used to associate test data with the code that it is meant to exercise.

issues

The techniques necessary to implement the Documentation Assistant are within reach of current technology. Related ideas for text representation and manipulation have been proposed elsewhere (e.g., [Nelson 82]). The challenge in building this type of system is providing the high degree of integration required for effective functioning, coupled with a simple interface for ease of use. Language-specific interfacing is necessary to provide support for linking code to documentation. Incorporating the system into existing environments may require a moderate amount of redesign in order to add the new capabilities. The system will fit nicely into the APSE environment, especially if an intelligent editor is eventually added.

A precursor to a documentation system is a set of definitions and requirements for different types of documentation for each phase of the software life cycle. The issues involved are not so much research issues as they are engineering (and, to some extent, management) issues. Efforts in this direction are currently being made, e.g., for test documentation [Gelperin 82].

The Documentation Assistant addresses many of the important problems associated with the documentation process. It cannot directly solve the major documentation problem, which is writing the documentation. The Assistant will require the programmer to spend more time thinking about and writing documentation, but this is desirable: it is the only way that good documentation will get written.

Documentation Assistant

time line

under 3 years

- collection of program comments as objects: The Documentation Assistant would collect documentation (as opposed to simply letting the programmer use a text editor to enter comments anywhere in the code), treating each piece of documentation specially, depending on its importance and relevance to the program.
- database for documentation collection: The collected documentation would reside in a database (such as the APSE database, which will be accessible to all APSE tools).
- simple dependency links: There would be links between the documentation and the program code, allowing the tools such as the Editor to display comments adjacent to the appropriate code.
- interface to source code control system: The documentation would be maintained using the same version control mechanism to be used on the program, thus allowing versions of documentation to correspond to versions of software.
- documentation guidelines: To effectively collect documentation, it is necessary to set up guidelines that specify what documentation is necessary, why it is necessary, how necessary it is, when to collect it, and where it should (logically) be placed (i.e., linked).
- information retrieval: An information retrieval system would allow documentation to be accessed independently of the Editor or any other development tools; this would be usable by those only trying to get information about the system (rather than changing it).

3 - 7 years

- documentation language: a language for representing documentation; the language would provide methods and constructs that are relevant to documentation (e.g., methods for talking about objects in general, rather than more specific constructs that talk about code).
- functional links between code and documentation: Links between documentation and code will specify the nature of the relationship (e.g., "this piece of documentation explains why the particular algorithm was chosen"); these links actually form part of the documentation.
- notification of out-of-date documentation: Using the version control mechanisms, it will be easy to spot documentation that refers to code that

Documentation Assistant

has been changed or deleted; various mechanism can be used to determine if the documentation should be changed (e.g., querying the programmer, semantic analysis of the code).

- graphical documentation: Since the documentation takes the form of a tree or a network, adequate presentation requires some way of graphically displaying the information.
- integrated with intelligent user interface: An intelligent user interface, such as the Intelligent Editor, would mask the Documentation Assistant; the programmer would only see an integrated programming environment which handles all aspects of the programming process.
- intelligent information retrieval: A rule based information retrieval system would allow flexible access, while requiring little knowledge about the structure of the documentation on the user's part; the retrieval system would use its knowledge about the structure of the documentation to facilitate the process.

over 7 years

- document interpretation: With tools that can perform semantic analysis on programs, the Documentation Assistant could interpret some of what the documentation meant, based on the interpretation of the code as well as the documentation formalisms provided by the documentation language.
- use of formal annotation language: A formal annotation language (section 9.6) used to describe programs more formally and completely can be interpreted and utilized as a form of documentation. Annotations would complement (and not replace) other forms of documentation.

Style Analyzer

9.4 STYLE ANALYZER

abstract

The Style Analyzer will check programs for adherence to predetermined standards and style guidelines. The rules used by the analyzer are independently specified, and will not be built into the analyzer; different users or groups of users can create their own standards.

need

Even the simplest programming languages have a good deal of flexibility, allowing programmers multiple ways of doing equivalent things. Programming languages also allow programmers to say things in distinctly awkward or confusing ways (e.g., the goto statement). These factors serve to decrease program readability and reliability. Programming language standards and style guidelines help minimize these problems, and should be part of every production programming environment. Style analyzers currently do exist, but they tend to suffer from lack of flexibility and poor user interfacing.

approach

Our approach to style analysis makes use of advanced techniques from language theory and artificial intelligence. The Style Analyzer will achieve its flexibility by separating the code that performs the analysis from the rules that specify what analysis to perform. This independence allows users to specify their own standards and guidelines without reprogramming the Analyzer, and is necessary for widespread acceptance, since many organizations have their own (differing) standards. Style rules can specify semantic constraints as well as the standard types of syntactic constraints that existing style analyzers enforce.

Human interfaces to style analyzers have tended to be awkward. First of all, style analyzers simply list all violations, leaving users on their own to fix all the problems. The proposed Style Analyzer could work in conjunction with the Intelligent Editor (section 9.2), allowing the user to either correct each style violation, or possibly tell the Editor to fix all occurrences of each violation (depending on the type of violation). Secondly, use of the APSE database provides a natural way of recording style information and associating it with code segments. After the first report of guideline violations, the programmer can specify that these violations should not be reported when the style analyzer is re-run. The violation information will still be available, but the programmer will no longer automatically be deluged with unnecessary violation reports.

Style Analyzer

use

The first step in deploying the Style Analyzer is the development of style guidelines. Guidelines can be developed at various levels in an organization, and so the Analyzer will be capable of handling a hierarchically organized set of rules. In actual use, the Analyzer can be invoked automatically by the Intelligent Editor or the Programming Manager, or the programmer can explicitly invoke it. The Analyzer will have no problems fitting into any environment (though an environment with capabilities similar to those of the APSE are preferable). The Intelligent Editor can provide a good user interface, but is not absolutely necessary.

examples

The following examples demonstrate some of the possibilities for style guidelines. Some of these are general, while other are aimed specifically at Ada. For the purposes of exposition, these guidelines are expressed in natural language; in a production system, they would be in a more restricted language.

formatting guidelines:

- blanks should be used around assignment operators
- loop bodies should be indented

syntactic guidelines:

- do not assign to loop variables inside a loop
- do not use nonportable constants
- do not mix positional and name notation in the same parameter list

semantic guidelines:

- do not use variables before they are set
- use enumeration types instead of integer types if no arithmetic operations are performed
- do not use in declarations expressions that may have side effects

Another example is a set of guidelines for choosing program identifiers, proposed by [Carter 82]. These rules help insure that names are readable, and

AD-A126 146

ADVANCED TOOLS FOR SOFTWARE MAINTENANCE(U) ADVANCED
INFORMATION AND DECISION SYSTEMS MOUNTAIN VIEW CA
J S DEAN ET AL. DEC 82 AI/DS-TR-3006-1 RADC-TR-82-313

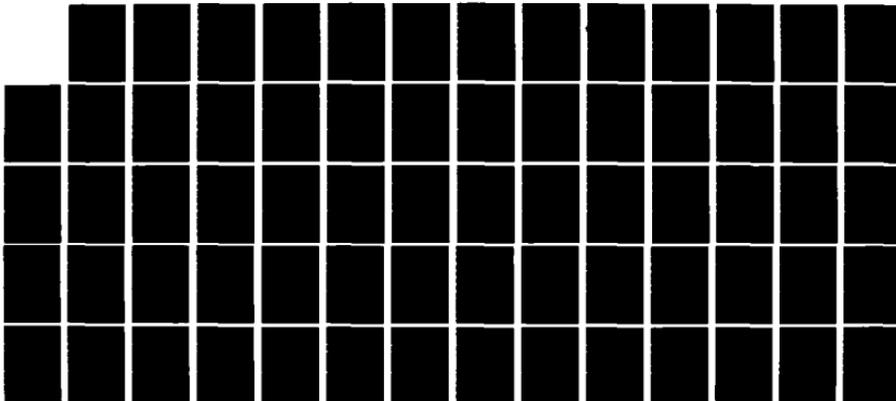
2/2

UNCLASSIFIED

F30602-80-C-0176

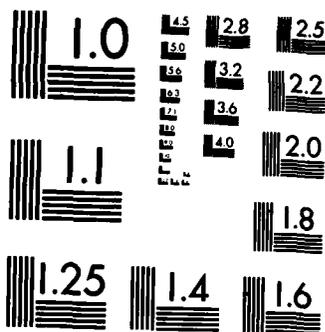
F/G 9/2

NL



END

FILMED
21
SERIAL
DTIC



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

Style Analyzer

well as insuring that names are selected in a consistent fashion.

issues

Some research will be necessary to develop a method for specifying style rules (something similar to a program reference language [Shapiro 82] may be one approach), as well as developing techniques for performing the analysis efficiently and incrementally. A style specification language allows the creation of style specifications which will serve as a style reference guide, as well as instructions for the style analyzer. Most importantly, it is necessary to develop style guidelines. Appendix A contains a sample set of style guidelines for Ada.

time line

under 3 years

- syntactic-based analysis: The Style Analyzer would enforce rules that can be specified in terms of the syntax of the language (e.g., "do not mix positional and names parameters in a parameter list [in Ada]").
- flow-based analysis: The Analyzer would enforce rules that can be specified in terms of a flow model (e.g., "complain about program statements not reached," "complain about variables never used").

3 - 7 years

- simple semantic analysis: The Analyzer would enforce rules that can be specified in terms of the semantics of the language (e.g., "do not build loops using a goto statement").
- simple style specification language: A technique for specifying style rules (independent of target language) would be used.
- rule-driven operation: The Style Analyzer would be driven off a set of rules representing the style guidelines for the target language.
- heuristic rules: Some style guidelines may be somewhat "fuzzy." Heuristic rules would allow these guidelines to be applied flexibly, by the association of variable weights with each rule (a technique used in other systems, such as information retrieval [McCune 82]).
- semi-automatic correction: In addition to specifying constraints, style rules could specify ways to fix code not conforming to the constraints (e.g., "loops built with a goto can be transformed (possibly with the help

Style Analyzer

of the Editor) into another type of loop").

- integrated with intelligent user interface: The Style Analyzer could make use of the features of the Editor (e.g., performing an analysis incrementally, making use of the Editor's ability to perform program transformations). The Editor could make use of style rules for creating code (and not just after-the-fact analysis).

over 7 years

- full style specification language: A complete style specification language would obviate the need to have any language or environment specific rules coded into the Analyzer.
- semantic analysis: The more thorough the semantic analysis of a program, the better the Analyzer will be about catching things that are really violations and ignoring things that are really not violations.

Metrics Tool Set

9.5 METRICS TOOL SET

abstract

The Metrics Tool Set will provide tools for measuring, analyzing, and assessing various properties of software systems. This information can be used for diagnostic and predictive purposes as well as for studying the evolution of a system.

need

Measuring aggregate properties of software can be particularly useful for software maintenance. This information can be used as a basis for decision making, ranging from lower level decisions, such as selecting a code change from a set of alternatives (based on the complexity of each change), to higher level decisions, such as deciding that a software system or subsystem needs to be rewritten (based on the complexity of modules and their interconnections). This information could also be used to identify weak areas or potential trouble spots of a system. In current programming environments, tools to extract this information are generally not available.

approach

The Metrics Tool Set will provide metrics to measure various aggregate software characteristics and, equally important, will provide support tools to help use the metrics intelligently and effectively. Metrics that might be included are operator/operand count metrics, control flow complexity metrics, interconnectivity metrics, etc. Support tools include collection tools to apply metrics (possibly automatically) and save the results in the APSE database, statistical tools for combining measurements and analyzing trends, graphics tools for displaying results of statistical analyses, etc. Individually, none of the tools is very complex; the power of the Metrics Tool Set is due primarily to the synergistic effect of an integrated set of tools (see section 8.2.1).

use

The Metrics Tool Set will be easy to use, and could be applied automatically by a tool like the Programming Manager (section 9.1). User education is an important factor in the effective use of the Tool Set; to a large extent, the value of statistics depend upon the skill of the interpreter. Statistics from various software systems, collected over time, will serve as a valuable reference point; but this database is not required for initial operation. As with the Style Analyzer, the Metrics Tool Set will fit easily into any environment, but an environment with the capabilities similar to an APSE would be preferable.

Metrics Tool Set

issues

The field of software metrics is an active research area (see, for example, [Perlis 81]). Designing and using metrics is still an art, rather than a science. However, even the use of simple metrics to help make simple decisions would still be better than nothing at all; there appears to be a real need for decision making aids in the maintenance process. The risk in building the Tool Set seems fairly low; more important is whether users will find it worth their while to use the tools.

[Kafura 81] cites four criteria for a practical metrics system:

- the metrics must be useable in a large-scale environment: they must be automatable, and it must be possible to take measurements during the design phase;
- the metrics should be complete and sensitive to changes in the structure of the system; they should also be sensitive to implicit data connections;
- the proposed metrics must be validated on actual large-scale systems;
- the measurements must be robust: they should apply to a wide class of structures, and should suggest causes for and solutions to structural deficiencies revealed by the metrics.

time line

under 3 years

- battery of metrics: The Metrics would use a set of metrics based on previous research.
- graphical statistics package: A statistical package would allow the analysis of trends based on data produced by the metrics; the need for graphical representation would be useful since the programmer would not have much assistance in interpreting the data (people better at detecting patterns when they are represented graphically).

3 - 7 years

- guidelines for metrics comparison: Guidelines would be set up for determining what metrics mean in terms of the overall software life cycle; methods for comparing different metrics would be established.

Metrics Tool Set

- simple trend interpretation: Techniques would be established for finding simple (e.g., linear) trends in the collected data (e.g., "the cost of making a single modification to a program appears proportional to a metric X").
- improved metrics: The set of metrics would be expanded for completeness; metrics that complement each other would be grouped together.
- validation of metrics: It would be necessary to determine which metrics are "meaningful," and which are useless or subsumed by other metrics.
- structural analysis based on simple semantic reasoning: Metrics will be more sophisticated, relying on semantic analysis of programs; these metrics would provide a "deeper" analysis, and would be more likely to measure the characteristics the programmer is really interested in measuring.
- sensitivity analysis: Sensitivity analyses would be performed for the metrics, providing a measurement of how metrics are affected by code changes.
- integrated with intelligent user interface: The Metrics Set would be integrated with an intelligent user interface, allowing the metrics to be invoked automatically on parts of a program, and allowing the results to be fed back into the programming process.

over 7 years

- metric-based acceptance testing: The Metrics would be used as an acceptance test; programs not satisfying the required criteria would be rejected (e.g., one acceptance criterion might be program complexity).
- trend interpretation: Analysis techniques would be used to create more complex interpretations of metrics (such as non-linear combinations of several metrics); these interpretations would be used to guide the maintenance process (e.g., determining the cost and ramifications of a software change on the basis of several metrics).

Annotation Language

9.6 ANNOTATION LANGUAGE

abstract

The Annotation Language will extend a programming language by allowing the programmer to state properties and aspects of programs that cannot be expressed in the programming language proper.

need

The Annotation Language is a vehicle for specifying intended program behavior. It is necessary for the formal, precise specification of program behavior, against which an actual program can be verified. Formal program verification augments existing debugging tools for applications where design correctness and reliability are particularly important. Viewed as documentation, program annotations also help to reduce the documentation problem.

approach

The Annotation Language combines elements of programming languages with formal logic. It can be thought of as extending a programming language with "formal comments." The Annotation Language may be regarded as a tool in itself, but it should be viewed primarily as the input to a system such as a formal program verifier. The verification system would perform the transformation of specifications and annotated programs into logical statements, which could then be proved (or disproved) with the help of automated reasoning aids (theorem provers).

use

The programmer will use the Annotation Language as a special type of programming language. Just what a programmer says with the Annotation Language will depend on why the annotation is being written; annotation information may be different for verification than for another purpose. The primary use of annotations is in program verification, but they are useful throughout the whole life cycle of software, especially if the various tools are sufficiently integrated and are able to interpret relevant parts of annotations. In compilation, annotations may express hints to the compiler for generation of optimized object code. In testing, annotations may be used to help generate test cases. Formal annotations improve documentation and maintainability; the judicious use of precise specifications of modules should increase programmer productivity.

Annotation Language

examples

Some example of Ada annotation are presented here; these examples come from [Krieg-Bruckner 80]. The annotations appear as Ada comments, prefaced by "--|".

```
subtype Even is Integer;  
--| for all X:Even --> X mod 2 = 0;
```

```
function Sqrt( N: Natural ) return Natural;  
--| return that S: Natural --> S**2 <= N < (S+1)**2;
```

```
procedure P( X: in Integer;           --| X > 0  
              Y,Z: in out Integer);  
--| where Y <= Z;
```

issues

Specification languages and formal verification are an important means for guaranteeing that a critical piece of software performs as expected. Formal verification remains a research area, but the technology is beginning to mature enough to move it into other environments, under carefully controlled conditions.

time line

under 3 years

- design of formal annotation language: An Annotation Language for the target language will be designed; this language will evolve as new uses for it arise.
- partial interpretation of annotation language: An analyzer would attempt to "understand" the annotations, making the information available to other tools (e.g., debuggers).
- limited verification: Based on current verification techniques, some verification of programs could be done, with the axioms stated as annotations.

Annotation Language

3 - 7 years

- extended multi-purpose language: The Annotation Language will be extended as more needs become apparent through actual use.
- extended verification capabilities: Continued research in formal verification would increase the sophistication of the verification tools.
- test data generation: Using annotations (which specify code behavior), test data could be automatically generated.

over 7 years

- full interpretation and verification: The eventual goal of the Annotation Language is to facilitate relatively "complete" verification and to allow semantic information derived from annotations to be fed back to other tools.

Change Propagation Detector

9.7 CHANGE PROPAGATION DETECTOR

abstract

The Change Propagation Detector (CPD) system will help analyze and isolate the effects of program changes.

need

When a change is made to a program, the effect of that change may propagate to other parts of the program. This often results in a "vicious" cycle where the programmer makes a change, tests the program, finds that a new bug has been introduced, so then makes another change, etc. The problem becomes larger when there is a long delay between introducing a bug and finding it; by that time, the program may have been released.

approach

The Change Propagation Detector will use knowledge of the syntax and semantics of the target language to determine the effects of change. Direct effects are easiest to find (e.g., if a change is made to a data structure, all direct references to the data structure can be tracked down). Indirect effects (e.g., variable references through pointers) and implicit effects (e.g., changes made to nonparameterized constants) can be more difficult to locate. The CPD system will make use of a variety of techniques for tracking down these changes. The system is meant to be flexible, allowing the addition of new techniques as they become available; due to the nature of the problem, it is unlikely that the system will ever stop growing. For example, as AI techniques mature, they will be added to the system to augment its capabilities. The CPD system should also be able to make use of "nonconventional" techniques. For example, it could make use of the dependency links provided by the Documentation Assistant (section 9.3) in order to locate indirect code dependencies. Once the system is capable of interacting with the Documentation Assistant, it could be used to detect the effect of changes on documentation. The CPD system may also be useful in other parts of the software life cycle, e.g., learning how programs work.

By design, the CPD system will generally take a conservative approach, warning about potential problems (since it cannot always distinguish between a real problem and a "non-problem"). The system can be incorporated into any programming environment, though an environment with capabilities similar to those of the APSE are preferable. Like the Style Analyzer (section 9.4), the CPD system will record its information in the APSE database, and interact with the user, possibly through the Intelligent Editor (section 9.2), to correct problems. When the user indicates a "non-problem", the appropriate database entry (e.g., the Diana tree [Intermetrics 81]) can be annotated, to avoid reporting the problem in the future.

Change Propagation Detector

use

The Change Propagation Detector will be used by the programmer during the process of modifying code. The programmer needs neither extensive knowledge of the target code nor of the tool itself. The CPD tool can easily be invoked automatically by other tools (e.g., the Programming Manager), and will fit nicely into any programming environment.

issues

The technology for analyzing data flow connectivity is available; more sophisticated checking, such as semantic checking, is still a research topic. The CPD tool is not a critical link in the maintenance process; its use is as an aid in preventing errors, but errors not caught will (should) eventually be caught during the testing phase. Recent programming languages provide facilities for modularity and data abstraction, which help reduce the problem to some extent, but they do not eliminate it.

time line

under 3 years

- syntactic and simple semantic detection: The Change Propagation Detector would use standard flow analysis techniques for tracking down the effects of changes.
- editor-assisted correction: The Detector would be interfaced to the Intelligent Editor (or some other user interface) so that all places needing modification (after a change) would be known to the Editor.

3 - 7 years

- detection and simple automatic correction: Effects of changes would be detected and automatically corrected if they matched a specific transformation rule known to the Detector (e.g., using a variable without declaring it could be fixed if the declaration could be inferred, or if the declaration was known elsewhere).
- simple inferencing: The Detector would use inferencing techniques to become better at really tracking down change propagation problems; inferencing would be based on a semantic analysis of the code.
- better techniques for detecting indirect propagation: The Detector would apply new techniques for tracking down indirect propagation, including inferencing, program annotations, and semi-formal documentation.

Change Propagation Detector

- integrated with intelligent user interface: The functions of the Detector would be available to the Editor or other interface, so that the programmer need not even be aware that propagation detection has been done.

over 7 years

- application domain knowledge: The use of knowledge about specific program applications would give the Detector additional capability for inferring the effect of a change.
- automatic detection and correction: The Detector would be able to find and explain (most) problems that resulted from a change to the code.

Test Case Analyzer

9.8 TEST CASE ANALYZER

abstract

The Test Case Analyzer will allow output of system testing to be automatically checked for correctness. It is based on a formalism for specifying correctness of system output and I/O relationships.

need

Over the life cycle of a piece of software, most testing activity is regression testing [Intermetrics 81]. Each cycle of regression tests generally involves a large number of test cases, resulting in voluminous output. The task of checking the output for correctness is tedious and error prone. Some programming projects have built systems that perform automatic checking, but these systems are special purpose, and are not easily adapted to other applications. Moreover, these systems usually perform very simple (e.g., bitwise) comparisons of the actual output with the expected output.

The standard method used in regression testing is to run the software on a set of test data and then do some simple comparison of the actual output with the expected output (often, the expected output is the actual output of a previous version of the software). The comparator may be a simple byte by byte comparator, or could be a more sophisticated differential file comparator (e.g., the Unix tool diff [Unix 80]). In either case, the comparator is ineffective when the concept of equivalence between actual and expected output is more sophisticated than a character by character comparison. For example, output may be considered equivalent if all printing characters correspond (i.e., "white space" is not significant), or if all corresponding numeric values are approximately equivalent (floating point calculations might have resulted in insignificant differences). In general, there is a need for the capability of specifying equivalence based on syntactic or semantic structure, in addition to lexical structure.

approach

The Test Case Analyzer will utilize more sophisticated methods for analyzing test run outputs, by providing a specification language for stating the correctness conditions for the actual test output, and an interpreter that checks the test case output with respect to the specifications. This allows much more flexibility in automating the testing process, by handling various kinds of outputs (including the types described above) without requiring programmer intervention. Reduced programmer intervention has important effects: more testing can be done, since the programmer does not have to monitor and examine the tests as carefully as before; errors are more likely to be caught, since more will be checked automatically, and since there will be fewer for the programmer to manually check; programmers do not need extensive

Test Case Analyzer

training to run the tests, since the analyzer relieves them of much of the work involved in analyzing the test results.

use

The major step in using the Test Case Analyzer is to write the output specifications; use of an Intelligent Editor (section 9.2) will ease this task. The tool itself can be invoked automatically by the test driver, leaving the programmer to simply examine the results. The Test Case Analyzer can fit easily into any programming environment.

The Test Case Analyzer is primarily directed at the testing phases of the software life cycle. Its use in testing can go beyond standard regression testing; for example, it could be used to check the results of instrumented code, insuring that no control flow anomalies have occurred. It does not attempt to address the issues of selecting test cases; rather, it helps apply tests correctly and efficiently. The Analyzer does have the capability for comparing any types of objects, and could be extended for use during other phases of the life cycle (e.g., determining if two versions of a program are syntactically equivalent).

issues

Some research will be necessary to develop a specification language and an interpreter. Building a general purpose specification language is an ambitious undertaking, but a conservative approach can make a positive impact without significant development risk. Unlike the Change Propagation Detector (section 9.7), the correctness of the Analyzer is very important, since it will perform the function of "certifying" test runs; if it functions improperly, software problems may not be detected during the testing phase.

time line

under 3 years

- diff: An existing tool such as the Unix tool diff would be used to compare output known to be correct with new output.
- syntactic equivalency specifications: The Test Case Analyzer would be extended to handle syntactic specifications of test case results.

Test Case Analyzer

3 - 7 years

- rule-based analysis: The programmer would be able to specify rules that would allow more flexibility than a context-free syntactic specification.
- simple output specification language: A language for specifying output correctness would be used, allowing the programmer to say how things should be, and letting the Analyzer figure out how to actually do the checking.

over 7 years

- knowledge-based representation of output specifications: Output specifications would be knowledge-based, allowing the semantics of the application to be used in analyzing the test case output.
- analysis driven by program specifications: Correct output behavior would be inferred from the specifications of the program itself, eliminating the need for test output specifications.

Intelligent Tutor

9.9 INTELLIGENT TUTOR

abstract

The Intelligent Tutor will help teach programmers about programming environments. It will always be available, not just during the learning phase, so programmers can learn or relearn as necessary at their convenience. It can be used to help experienced programmers adapt to a new environment as well as to teach new programmers.

need

The training of personnel is a major problem during the software maintenance phase. Programmers must not only learn the programming environment; they must also become conversant with the system to be maintained. There are virtually no tools in existing or proposed programming environments that address this problem. At best, existing environments provide some sort of on-line manual with a flexible access method. None help teach; users are expected to have enough knowledge from the start.

approach

The key to building a useful computer-assisted instruction (CAI) system is to build "intelligence" into the system. Many existing CAI systems provide little more than computerized page turning and test scoring. The Intelligent Tutor for the APSE environment will utilize knowledge about the programming environment (both Ada and the APSE) as well as knowledge about programmers themselves, in order to provide a sophisticated form of online instruction. The Tutor's knowledge about the environment will allow it to use the environment as well as to teach about it. For example, it might ask the student to write a program to perform some function. In order to check the program, the Tutor can compile and run the program, checking at all stages of the process to ensure that there are no program errors.

use

The Tutor will be capable of working in several modes. While the user is learning, the system will ask questions, and the user will answer. When the user is actually programming, s/he can turn around and ask the system questions. This is referred to as a mixed initiative scheme, and allows enough flexibility to be used by experienced as well as novice users. In system initiative mode, the Tutor simply follows some predefined curriculum, with variations based on the user's capabilities.

In user initiative mode, the Tutor must be able to answer questions as well as make explanations. The following (somewhat arbitrary) distinction is made

Intelligent Tutor

here between question answering and explanation: question answering involves looking up answers (e.g., in a database) or applying a priori rules to determine the answer (e.g., running the compiler on a program to check for syntactic errors); explanation, on the other hand, requires more reasoning capabilities, allowing the Tutor respond to questions like "Why did this happen?" and "What will this do?". Eventually, the Tutor should be able to critique programs (making use of other tools in the environment to perform various decompositions and analyses) and help users write programs.

The Tutor will easily fit into any programming environment, though an environment with capabilities similar to the APSE is preferable. Its intelligence may make it invaluable for many (all?) phases of the software life cycle. It might be considered to be an intelligent help system, or, maybe, like a friendly expert programmer who has also memorized all the manuals.

issues

The CAI field has been around for quite some time, and so a straightforward tutoring system is easily within reach. Incorporating intelligence into a CAI system is still a research area, but the necessary techniques are indeed being developed (see [Gable 80] for a survey of AI techniques in CAI). The more advanced aspects of the Tutor, such as explanation and critiquing, still need more research. Another area important to the success of the Tutor, and still undergoing research, is knowledge representation and acquisition (e.g., [Wescourt 77]). To make the Tutor general and useful, it is desirable to use an explicit representation for knowledge, rather than hardwiring it into the code (an approach sure to guarantee quick obsolescence).

The risk in building a full-scale Intelligent Tutor is relatively high; such systems in the past have rarely made it out of research labs. However, parts can be built without much risk. Its reliability is important, since programmers will make decisions based on it, but, as its effect on target software is indirect, there is not much risk in using the tool (compare it to a manual or teacher with erroneous information).

The use of the Tutor will have a significant impact on the maintenance process, for two reasons. First, due to high turnover of personnel, there is a need for continual retraining in each organization. Since the Tutor will be able to teach specific systems as well as the general APSE environment, each organization can use the Tutor to teach whatever is necessary to begin work in a new environment. Second, even without rapid personnel turnover, there is a continuing need to train people, especially with a new programming environment such as the APSE.

[Gable 80] lists qualities desirable in a human tutor; these are reproduced below as guidelines for what an Intelligent Tutor might strive to do.

Intelligent Tutor

1. The tutor causes the problem solving heuristics of the student to converge to those of the tutor.
2. The tutor will learn and adopt student solution methods if they are superior.
3. The tutor chooses appropriate examples and problems for the student.
4. When the student needs help, the tutor can recommend solution scheme choices and demonstrate how to apply techniques.
5. The tutor can work arbitrary examples chosen by the student.
6. The tutor is able to adjust to different student backgrounds.
7. The tutor is able to measure the student's progress.
8. The tutor can review previously learned material with the student as the need arises.
9. The tutor will give immediate feedback on errors while allowing the student a free hand in deciding how to solve a problem.
10. After the student solves a problem, the tutor may point out more direct solutions or ones that use recently learned theorems or techniques.

time line

under 3 years

- simple instruction, independent of the compiler: The Intelligent Tutor would use prepared materials like conventional CAI systems.
- script-driven lessons, with simple heuristics: The Tutor would make use scripts outlining different possible user responses, with methods for trying to be intelligent about what to do.
- intelligent information retrieval: This would be a capability for the user to use the Tutor directly as a source of information, without having to use a formal information retrieval query language.

3 - 7 years

- simple question/answer capability: The Tutor would be able to answer questions, using some combination of information retrieval, scripts, and inferencing techniques.

Intelligent Tutor

- rudimentary knowledge of language and environment: The Tutor would have a knowledge base and inferencing capability that would allow it to deduce information that was not explicitly given to it.

over 7 years

- program analysis and critique: The Tutor would be able to take programs written by the user, analyze them, and critique them; the critique would cover style and structure as well as logical correctness.
- modelling of human users: The Tutor would use knowledge about how people learn to understand why the user was having difficulty understanding concepts.
- error explanation: The Tutor would detect errors made by the user, and then explain the error to the user; it might also point out how to fix the error, and ways of avoiding it in the future.

10. EVALUATING ADVANCED SOFTWARE MAINTENANCE TOOLS

An evaluation of the tools proposed in the previous section was undertaken. This evaluation was done to confirm our judgment that these nine tools were in fact feasible and desirable, to sharpen our intuitions about their prospects and potential problems, and to aid in deciding the relative importance of the tools (and hence the recommended order of development). The evaluation was guided by AI&DS staff; the evaluators consisted of both Air Force personnel and AI&DS staff. While only the proposed tools were evaluated, the evaluation techniques are general enough to be applied to other tools also.

10.1 TECHNIQUES FOR EVALUATION

One of the goals of this project was to develop criteria and metrics by which tools can be evaluated for their impact on software maintenance problems. Unfortunately, the task of developing criteria and metrics is quite difficult. A metric deemed "usable" by one person is often deemed "unusable" by another. Attempts to quantify questions like "...what tool is best?" are often controversial. Several approaches were considered, including an informal empirical approach and a more formal decision-analytic approach. An empirical approach was chosen. While lacking the rigor of a formal mathematical model, this approach seems adequate in the context of this project.

Based on interviews with Air Force maintenance personnel, as well as a review of the literature, a set of questionnaires was developed to measure the utility of maintenance tools. The questionnaires were administered in two rounds, to both Air Force personnel and AI&DS staff. The set of people answering the first round was not identical to the set answering the second round (but there was some overlap). The first round assessed the importance of each of the criteria in the questionnaire. The second round evaluated each of the tools with respect to these criteria.

Most of the criteria on the questionnaire were derived from the criteria specified in the Statement of Work for this project. These criteria are:

- potential impact on software maintenance problems, i.e., which functions of the maintenance process will be reduced, and to what degree;
- feasibility of application and implementation in a "production" and maintenance environment;
- estimated development cost;
- computer resources required for recurrent use;

Techniques for Evaluation

- prerequisite level of user education and training;
- relationship with common software development and maintenance tools.

10.1.1 Evaluating Criteria

The questionnaire for the first round was designed to evaluate the importance of various criteria against which tools could be evaluated. The results were meant to be used to evaluate the results of the following round. Hence, the results of this round could be considered an intermediate product.

The questionnaire was divided into four categories: feasibility, capabilities and benefits, costs, and qualities. Each section contained a list of criteria that could be used to evaluate a tool. The criteria were defined by AI&DS; the list is reproduced in Appendix C. Respondents were asked to rate the importance of each criterion on an integer scale from 0 to 10, with 0 signifying "the criterion is of no importance" and 10 signifying "the criterion is of extreme importance".

This questionnaire was sent out to four Air Force sites (Rome Air Development Center, plus the three sites described in section 4.1: SCF, CCPC, and SAC), and was also filled in by members of the AI&DS staff. A total of thirty questionnaires were completed (twenty-six by Air Force personnel and four by AI&DS personnel).

To evaluate the questionnaire, the rankings for each criterion were averaged, producing a weight for each criterion which reflects the importance of that criterion. The weights were designed to be used during the second round of questionnaires, and so are not of direct interest here (for reference, they are included in Appendix C, listed next to the criteria). The weights were examined to ensure that all of the evaluation criteria were useful (and they were: There was no criterion that was uniformly rated low). In computing the weights, the evaluations from all sites were lumped together; no attempt was made to create separate sets of weightings for each group of evaluators.

10.1.2 Evaluating Tools

The second round of questionnaires was designed to evaluate individual tools. The questions were the same as the questions in the first round, but instead of asking "How important is this criterion?", the question was "How does each tool rate on this criterion?".

Since the tools being evaluated were designed by AI&DS staff members, the questionnaires were split into two sets. One set of questionnaires (see Appendix D) had criteria best evaluated by potential users of the tool; this

Techniques for Evaluation

was the set used by Air Force evaluators. The other set (see Appendix E) contained criteria best evaluated by those knowledgeable about the design and implementation of the tool; this was the set used by AI&DS evaluators. The questions in the two different sets were mutually exclusive (with two exceptions, discussed later).

Along with the questionnaire, evaluators were given a draft copy of the previous chapter of this report, which describes the tools to be evaluated. They were asked to rate each tool on all criteria. Since nine tools were to be evaluated, considerably more work was required for the evaluation. Hence, the number of evaluators was reduced to thirteen (eleven from the Air Force and two from AI&DS).

There were several steps in evaluating these questionnaires. First, the answers were averaged across respondents, resulting in nine sets of averages, one set for each tool. Then, each set of averages was multiplied by the previously collected weights, giving a set of weighted averages. Next, each set of weighted averages was totalled, resulting in nine scores, one for each tool. The scores, in descending order, are presented in terms of percentage of the maximum possible score:

Composite Ranking of Usefulness

<u>score</u>	<u>tool</u>
54.8	Documentation Assistant
52.8	Metrics Set
51.8	Programming Manager
50.1	Change Propagation Detector
49.2	Intelligent Tutor
49.1	Style Analyzer
49.1	Test Case Analyzer
48.3	Intelligent Editor
40.8	Annotation Language

The ranking of these tools provides some insight into how the evaluators felt the tools satisfied the selected criteria. Each tool was independently ranked on an absolute scale for each criterion. These values were used to form the scores shown above. However, we feel that the most appropriate way to interpret the tool ranking implicit in the ordered list of tools above is to pay attention to the relative ordering of tools and to the difference between tools; the specific score is much less important. This ranking provides a relative evaluation. A low ranking does not imply a tool is worthless; rather, it means that the tool appears less useful than the other tools being evaluated. For more detail, see Appendix F, which presents scores broken up by questionnaire category and respondent.

Techniques for Evaluation

Two extra questions were added to both sets of questionnaires in the second round. These questions asked respondents to rate each tool on its "overall practical usefulness" and "overall theoretical usefulness." Practical usefulness refers to the usefulness of the tool, given real-world constraints in building and using the tool; theoretical usefulness assumes that building and deploying the tool is not a problem. The purpose of these questions was to allow a simple, direct evaluation of each tool. The results below from these questions are simply averaged responses. The results, in descending order, were as follows:

Overall Practical Usefulness

<u>score</u>	<u>tool</u>
6.0	Documentation Assistant
5.8	Metrics Set
5.5	Programming Manager
5.4	Style Analyzer
5.3	Change Propagation Detector
5.3	Intelligent Tutor
5.1	Intelligent Editor
4.7	Test Case Analyzer
2.8	Annotation Language

Overall Theoretical Usefulness

<u>score</u>	<u>tool</u>
8.0	Programming Manager
7.9	Intelligent Tutor
7.6	Documentation Assistant
6.8	Intelligent Editor
6.8	Style Analyzer
6.8	Metrics Set
6.8	Change Propagation Detector
6.8	Test Case Analyzer
6.3	Annotation Language

These simple rankings were meant to provide a more intuitive way of evaluating tools, and should be compared with the more complex ratings to provide additional support for the evaluation techniques. Two tools, the Documentation Assistant and the Programming Manager, fell in the top three for all rankings. These tools appear to be excellent candidates for further study and implementation. Two other tools, the Metrics Set and the Change Propagation Detector, ranked fairly well except on the theoretical usefulness question. Since these tools are simpler than many of the others, they probably

Techniques for Evaluation

ranked high because they are feasible in the short term. Their lower theoretical ranking most likely indicates that their long term potential is not as great as some other tools.

10.2 TOOL RATINGS - MAINTENANCE PROBLEM AREAS

Another way of looking at tools is in terms of how they address the major maintenance problems (as identified in section 4.2.3). Table 10-1 identifies each tool's ability to deal with the various maintenance problems. The maintenance problems are listed on the top of the table; they appear in descending order of importance.

Not surprisingly, the two top rated tools from the previous section (the Documentation Assistant and the Programming Manager) both address all of the most important problem areas. Note these ratings indicate what problems each tool addresses; they do not indicate how well the problem is addressed.

10.3 SUMMARY

The results reported in this chapter were meant to provide a method for evaluating software maintenance tools. The evaluation techniques are generally applicable, and could easily be applied to other tools. With repeated use, norms could be established, allowing tools to be evaluated on an individual basis.

The proposed tools were chosen on the basis of real problems and perceived needs. Thus, all of the tools are likely to have a large positive impact on the maintenance process. Even low ranking tools would be likely to provide significant benefit.

PROBLEM AREA

TOOL	personnel turnover	poor documentation	making changes	diagnosing/ monitoring
Programming Manager	++	++	++	++
Intelligent Editor	+		++	
Documentation Assistant	+	++	+	+
Style Analyzer	+	+	++	
Metrics Set			++	+
Annotation Language		++	++	+
Change Detector		++	++	
Test Case Analyzer	+	+	++	
Intelligent Tutor	++	+		

Legend

- ++ the tool directly addresses the problem
- + the tool indirectly addresses the problem
- blank entry means tool does not address problem

Table 10-1: Rating of Tools with Respect to Problem Areas

11. CONCLUSIONS

11.1 ACCOMPLISHMENTS

11.1.1 Identification of Major Maintenance Problems

Four major problems have been identified in the software maintenance process. They are:

1. high turnover of personnel and lack of qualified replacements
2. understanding what a software system does, without good documentation
3. determining all relevant places to make changes
4. diagnosing and monitoring operations

A major factor in all of these problems is the difficulty people have in comprehending complex software systems. If one problem were to be selected as the most significant maintenance problem, it would be the problem of comprehension. This conclusion differs from the widely held belief that ease of making changes is the most important factor in software maintenance.

11.1.2 Identification of Useful Tools

After identifying the major maintenance problems, a number of tools capable of making a large impact on these problems were proposed. These tools are summarized below:

- The Programming Manager assists the programmer by systematically applying administrative and technical policies, as well as helping apply both general and application-specific programming techniques and methods.
- The Intelligent Editor provides facilities for manipulating programs at several conceptual levels (e.g., textual, syntactic, semantic, and intentional), and provides an intelligent interface to other tools.
- The Documentation Assistant is a tightly woven collection of tools for creating, structuring, maintaining, and accessing all forms of documentation.
- The Style Analyzer checks programs for adherence to programming standards and style guidelines (which are expressed with a specification method that is independent of the analysis process itself).

Accomplishments

- The Metrics Tool Set provides tools for measuring, analyzing, and assessing various properties of software systems over their lifetime.
- The Annotation Language is a method for extending a programming language by allowing annotations which specify state properties and other aspects of programs that cannot be conveniently expressed in the programming language itself.
- The Change Propagation Detector analyzes a program for effects of a program change.
- The Test Case Analyzer allows the output produced by test runs to be automatically checked for correctness, based on a formal (or informal) specification of what the output should look like.
- The Intelligent Tutor uses a knowledge-based approach to teach programmers about programming languages and programming environments, using the tools themselves.

11.2 RECOMMENDATIONS

11.2.1 Proposed Tools

All nine tools should be developed and used in the Ada Programming Support Environment.

Looking at long term solutions, three of the proposed tools seem particularly valuable, because they provide significant capabilities in all phases of the maintenance process. The Programming Manager presents a new approach to an old problem: How can people work more effectively in a complex environment? In a software maintenance environment, complexity can be introduced in many places: the applications software, the tools used for performing the maintenance, and the administrative environment which controls the process. The Programming Manager provides a mechanism for guiding the programmer through this maze. The Documentation Assistant provides an innovative solution to the problem of documentation. By recognizing that documentation can be an entity unto itself and by providing tools that handle those unique requirements, the whole documentation process can be improved. The Intelligent Editor itself is not a new idea; however, it goes significantly beyond current efforts by providing a great deal of sophistication in the manipulation of programs and software systems. While all these tools may be long term efforts, each can be scaled down to provide similar capabilities for shorter term goals, particularly the Documentation Assistant.

Several of the proposed tools are less comprehensive, attempting to solve smaller problems. These tools provide capabilities that may already be available, but they also employ advanced techniques which provide much greater

Recommendations

depth and sophistication than existing tools. Style analysis is not a new problem, but the proposed Style Analyzer provides new capabilities by allowing the style rules to be specified independently of the analyzer (most style analyzers have the guidelines coded in-line). Software metrics are a somewhat newer phenomena. The Metrics Tool Set provides more than just metrics: it provides tools for analyzing and saving metrics. The Change Propagation Detector provides capabilities similar to that of a data flow analyzer; it extends these capabilities by attempting to detect implicit flows, using information not generally available to a data flow analyzer (e.g., program assertions). This also allows it to avoid the "pessimism" typical of analyzers which make worst case assumptions when in doubt. The Test Case Analyzer automates the tedious and error-prone task of checking the results of test runs. Existing tools of this type tend to be project specific. The automated approach allows the expected outputs to be formally (or informally) specified, thus eliminating the need to build new special purpose test analyzers.

The effectiveness of these simpler tools can be greatly enhanced by the use of the Intelligent Editor. The Editor provides a uniform interface, and allows the programmer to converse at a higher level, since the Editor can make the mapping between high level statements and the actual code. For example, a programmer might ask it to perform a style analysis on all loops containing multiple exits. The Editor is capable of figuring out which parts of the program satisfy this condition, and can then pass the appropriate statements to the Style Analyzer.

Two tools require separate treatment. The Annotation Language is actually more of a technique than a tool. This probably explains why it ranked lowest in all the evaluations. The technique of annotating programs, which arose from research in formal program verification, is still relatively new. Its full potential has not yet been uncovered. The Intelligent Tutor, on other hand, provides a function which has been around for a while: computer assisted instruction. The Tutor takes an unconventional approach by incorporating knowledge into the system. This includes knowledge about the human student, as well as knowledge about the programming domain. There are two somewhat separate tasks required to build an Intelligent Tutor. Mechanisms to manipulate the knowledge and interact with the user need to be constructed. Equally important is the development of a curriculum and courses. While much of the former effort can be considered longer term (due to the necessary research), the latter effort is short term.

11.2.2 Development Strategy for Proposed Tools

For greatest reward, we recommend three tools to be developed as part of a medium term effort (3 - 7 years): the Programming Manager, the Documentation Assistant, and the Intelligent Editor. However, all of the proposed tools incorporate unique or untried solutions to various maintenance problems, and therefore justify further research and development.

Recommendations

These selections were made on the basis of our expert opinion, influenced by the tool evaluations discussed in the previous chapter. We emphasize tools that will provide long term benefits and solutions, over tools that are more immediately realizable but have less to offer in the long term. Both the Documentation Assistant and the Programming Manager provide long term solutions to many of the major maintenance problems (see 10.2). Both were highly rated in the evaluation process. The Intelligent Editor, though not rated as highly, was selected because of its ability to act as an interface for a number of the proposed tools, including the Documentation Assistant. Accordingly, its initial development should be focused on its ability to serve as a common interface to other tools.

For a shorter term effort (under 3 years), we recommend simpler versions of the Documentation Assistant and the Metrics Tool Set. These two tools were ranked at the top in terms of immediate practical benefit. We feel that a useful version of each could be in use within three years of the start of a tool development effort.

11.2.3 Other Research Areas

In studying maintenance problems and designing tools to solve them, we have identified important research topics that warrant further investigation, either as independent inquiries or in conjunction with tool development. The topics fall mainly into the categories of software engineering and artificial intelligence. Because of the sophistication of the proposed tools, it is not surprising that some of the topics seem to fall into both categories. The research topics most relevant to the proposed tools are listed below:

- software methodologies: Promising development and maintenance methodologies need to be identified and further refined. The Ada effort has led to a standard language, and a possible standard environment, but no methodology. The Programming Manager does not require a methodology, but if one is used, the Manager can help in its use.
- metrics: Much work has been done on developing metrics; little has been done on using them. Interpretation and application of program metrics is still not well understood.
- specification techniques: Several of the proposed tools rely on formal specifications for their operation, including the Annotation Language (for program annotations or formal comments), Style Analyzer (for guidelines) and the Test Case Analyzer (for expected outputs). Specification techniques include both representational issues as well as application issues.
- reasoning about programs: To aid the programmer, it may be necessary to understand the programs that are being manipulated. For example, the Intelligent Editor should allow the programmer to talk about a program in

Recommendations

terms of what it does; the Style Analyzer needs to make use of information which goes beyond that directly derivable from the source code.

- reasoning about programming: To help the programmer make decisions about what to do, the Programming Manager needs to understand the programming process itself. The Intelligent Tutor needs to reason about the programming process in order to understand what the programmer is doing wrong (or if the programmer is doing things correctly).
- reasoning about programmers: The Programming Manager cannot effectively aid the programmer unless it has a good model of what help the programmer needs, and the ability to determine when to help out. The Intelligent Tutor also needs to have a model of how people learn.
- friendly user interfaces: Many of the tools converse with the programmer. To be used effectively, it is necessary for the user to understand what the tool says, and how to respond to it.
- multiple users and knowledge bases: Tools such as the Programming Manager must coordinate the activities of many programmers working on different parts of one project (or even on separate projects) over long periods of time. This is significantly harder than managing the activities of just one programmer.
- information structuring: How should information about a program be structured so that it is understandable by people? The Documentation Assistant collects large quantities of information; however, it can't simply throw the information at the user. It should understand that information must be presented selectively, to avoid overloading and confusing the user.
- evaluating tools: The work on evaluating tools should be extended. Techniques must be developed to evaluate tools after they are deployed, to determine how well they really work, and how they can be improved.
- understanding software maintenance: More data on software maintenance activities is needed to allow the development of a comprehensive process model of software maintenance. This would help in understanding where the bottlenecks are. Such an understanding would point the way to administrative and methodological improvements, as well as identify critical needs that could be addressed by maintenance tools.

11.2.4 Other Recommendations

There are many other comments that could be made about software maintenance practices in the Air Force. Since these comments are peripheral to the main focus of our work, we mention only a few key ones here.

We believe that SAC's strategy of keeping maintenance personnel for a tour of

Recommendations

duty that is longer than normal is a good idea. This allows for more complete training at the start of the tour and a larger return on that investment thereafter.

Large C3I systems that include computer hardware and software must be procured using more realistic estimates of cost and life span. If a system's life span is estimated to be upwards of a decade during requirements analysis, then the appropriate magnitude of investment during the development phase will be much clearer. Specifically, systems should be designed and documented with maintenance and long life in mind.

Although this is contrary to the recommendations of some, we feel that it is not a good organizational or technical practice to separate development and maintenance programmers, particularly within the same project or system. This immediately leads to a loss of esteem for the maintenance staff.

11.3 DEPLOYING NEW TOOLS

Incorporating a new tool into an existing environment requires more work than simply installing the tool. Before installation, one must ensure that the tool is compatible, in some sense, with the environment, and there must be a need for it (as a far fetched example, a Fortran preprocessor doesn't make much sense in a Jovial environment). Once installed, users have to be identified and trained. Also, changes may have to be made to the environment or to the applications software.

11.3.1 The Ada Programming Support Environment

This study has assumed that the Ada Programming Support Environment will be the target environment for all the tools proposed here. The APSE will contain features used in some of the most successful programming environments. Incorporating sophisticated tools into the APSE will be easy; many of the necessary features, considered fancy in some systems, are standard in the APSE (for example, a central database for storage of historical information).

11.3.2 Retrofitting New Tools to Old Applications

Once a new tool is installed, any compatibility problems between the tool and the applications software must be worked out (e.g., a tool might require that code be in a certain format, or might require that a certain development methodology was used). Of the proposed tools, some are simple enough to be used with no compatibility problem. For example, the Intelligent Editor, the Test Case Analyzer, and the Change Propagation Detector can all be used with little or no adaptation.

Deploying New Tools

Other tools, which capture "historical" information during the development and maintenance phases, are somewhat more difficult to incorporate. For example, the Metrics Tool Set collects measurements over the lifetime of a software system. If measurements are missing for part of the tool's lifetime, initial use of the tool may be somewhat frustrating, since it can collect data but not produce much in the way of analysis. This is the initial build-up period, where the tool may be "consuming" more than it is producing. Other tools of this type are the Documentation Assistant and, to some extent, the Programming Manager, since they collect information during the entire development and maintenance process. It is important to recognize and plan for this initiation period before installing a tool of this type.

Another class of tools is that which requires additional (non-historical) information. For example, to use the Annotation Language, programs must be annotated. However, it doesn't matter when the programs are annotated; they can be annotated when they are first written, or can be annotated later. Annotations are not considered historical information because when the code changes, the annotations change too; the annotations never need to refer back to older annotations. Similarly, the Style Analyzer can be run on any program. However, programs developed without the aid of an analyzer will require more work if they need to be changed to conform to standards.

Finally, there are tools that utilize knowledge about their environment. For characteristics that are invariant across environments (e.g., all Air Force sites will use the Ada language), no special effort is necessary. However, for characteristics that do vary with the environment, these tools will need to have their knowledge bases updated. Both the Intelligent Tutor and the Programming Manager fall into this category. If necessary, both can function with incomplete environment-specific knowledge, but then, of course, their capabilities are not fully realized.

11.3.3 Training People to Use New Tools

Installing a new tool requires more than simply making it available. People have to be trained how and when to use it. This is basically an administrative problem. However, two of the proposed tools can aid the training process. One is the Intelligent Tutor. Once a "course" has been developed, it can be made available to all who have access to the Tutor. "Classes" are individualized, and are held at the convenience of the individual. Another helpful tool is the Programming Manager. As mentioned above, people must be taught when to use the tool. For some tools, of course, this is not a real problem; i.e., all programmers know when to run a compiler. But other tools, such as a style analyzer or runtime monitor, are more difficult to handle. The reasons for invoking these tools may be technical (e.g., to catch more program errors more readily) or administrative (e.g., to ensure that all code conforms to standards). The Programming Manager maintains this type of knowledge, and assists the programmer in making the appropriate decision. Even if a new tool were added by telling only the Programming Manager and the Intelligent Tutor

Deploying New Tools

(and not the programmer), it would be possible for the tool to be used appropriately and effectively. If a situation arose where the tool was applicable, the Manager would mention it to the programmer, who would then realize there was a new tool. At that point, depending on the tool, the programmer could: go ahead and use it (if he were familiar with that type of tool); invoke the Tutor to learn about the tool; or ask the Manager to invoke the tool automatically.

APPENDIX A. ADA STYLE GUIDELINES

This appendix contains some sample style guidelines for the Ada language. The style analyzer described in section 9.4 would make use of a set of rules such as these. Note that these rules are in no way meant to be complete or definitive. Some of these guidelines are easily checked automatically, while others would require considerable work to perform automatically.

This list was compiled by Prof. David Luckham of Stanford University as part of this project.

SOME ADA STYLE GUIDELINES

David C. Luckham

February 1982

At the present time there is not enough experience with Ada to really compile a comprehensive list of Do's and Don't's. The lack of compilers at present means that most Ada "experience" comes from other Algol-like languages. There is very little experience with compilation and running of tasking programs.

The following list of specific style recommendations comes from some classroom experience teaching Ada, and some limited experience compiling and running Ada programs using the Adam compiler at Stanford University. It is expected that this list will be greatly expanded over the next year or two, and that some items will be revised. Those items that could be enforced by a checking tool and would be worth enforcing in some applications programs, are marked with a comment preceded by "***". The categories and top-level category numbers below correspond to the chapters of the Ada manual [Ada 80].

2. Text Conventions

- .1 Stick rigidly to formatting conventions -- Readability.

Also: develop conventions for situations where the Ada text overflows a line.

For example, in the case of long subprogram parameter lists, each sublist of a given type could be placed on a separate line, the set of lines being delimited by the parentheses opening and closing the parameter list.

*** This can be checked. Formatting conventions may vary from installation to installation as a matter of taste. Format checkers would be easy to build.

- .2 Use distinct identifiers for formal parameters of subprograms in same scope -- To avoid naming confusions.

***Checkers for formal name clashes may be very well worthwhile, especially on large projects, since such clashes may indicate some more basic misunderstandings between programmers.

- .3 Do not mix positional and named notation in the same parameter list -- Readability.

Although Ada permits this, it seems to introduce more confusion;

when in doubt use the named parameter form for clarity.
*** This would be checked by format checkers for 2.1.

.4 Mnemonic names for all identifiers: verbs for subprograms, nouns for objects, types, and packages.

.5 Blanks should be used after ",", and before and after ":", ":", and ">".

.6 Use blank lines to delimit related groups of declarations within a declarative part; bodies of program units should always be in the same order as the corresponding specifications.

3. Types

.1 Make strongest possible use of strong typing: e.g., different types should be declared for sets of objects if there is no common processing between objects in the different sets.

*** Analysis for groups of independent objects of the same type is probably time-consuming but possible. Strategies for where to check for this need to be worked out if the results are to be useful. E.g., redeclaring new types for independent sets of objects should be done in basic, most widely used modules of a system; this ensures that later use will not confuse those independencies.

.2 Derived types should be used in preference to subtypes to express logical independence.

However it should be noted that the inheritance of subprograms by a derived type from its parent type is poorly defined. As a general style rule the parent type should be encapsulated with its selectors and constructors (subprograms to be inherited) in a package specification to clarify exactly what is inherited. Derivation (declaration of a derived type) should never be in the body of that package encapsulating the parent type.

.3 Enumeration types should be used instead of integer types if no arithmetical operations are performed.

.4 Always use default initial values for discriminants of variant records

-- This avoids constraints on declaration of variant objects which then fix the variant part of the object thereby not permitting the variant part to be changed. Essentially, this use of a variant record type is equivalent to a union of disjoint subtypes.

(a) If a variant record type is declared without default initial values for the discriminants, then a further set of subtypes corresponding to the cases of the variant parts should be declared -- this makes an explicit declaration of the actual use of the variant type.

4. Expressions and Overloading

- .1 Try to use static expressions wherever possible in declarative parts.
- .2 Avoid expressions in declarations which, during elaboration, may have side effects on other objects than the one being elaborated.

5. Statements

- .1 Loops should not be built with "go to", nor should exceptions be used for the normal exit from a loop.
- .2 Loop exit: place "exit" clearly at top level of loop body -- use "when" rather than nesting in "if else".
- .3 A Loop with more than one exit should be carefully analyzed with a view to clarifying its functionality; e.g., Can it be separated into two or more one-exit loops? Can it be abstracted out as a subprogram?

6. Subprograms

- .1 If a subprogram propagates an exception, that fact should be documented by means of a comment in the specification of the subprogram. -- A compilation unit should have "no surprises".
*** A tool to add such declarations to subprogram specifications can be built, and would be a useful part of an automatic documentation system.
- .2 If a set of subprograms share a set of global variables, then the subprograms and common globals should be encapsulated in a package body. The visible part of the package should contain the subprogram specifications.
- .3 Subprograms should have small bodies in general, say on the order of 10 Ada statements. Large subprograms should be analyzed with a view to "packaging" into smaller units, especially when that subprogram may be shared in multitask computations.

7. Packages

- .1 Despite Ada visibility, packages should be treated as closed scopes with respect to objects.
The use of outside global units in packages should be restricted to context specifications.
*** Tools to check this can be built and could be very important in checking secure software, and in maintenance. Such "Imports" checkers could either add "imports declarations" to package

specifications or bodies, or give warnings.

- .2 Use private types in visible specifications wherever possible in place of complete type declarations.
- The effect of a change in type implementation is then localized to the body of the exporting package.

7A. Documentation of Packages

.1 Package visible parts should declare all exceptions propagated from the package operations.

*** As before in 6.1.

.2 Normal Ada visibility rules should be observed in the comments; thus, implementation details hidden in a package body may not be referenced in comments in the package visible part.

.3 Ada conventions with respect to repetition of subprogram specifications in the corresponding bodies should be observed also for comments documenting subprogram specifications.

9. Tasks

.1 Declare all task types at outermost level (if possible)

-- clarity of program and facilitation of analysis for liveness errors and resource requirements. Also lessens need for use of task TERMINATE construct.

.2 Do not share globals between tasks if possible.

Stick to the recommended Ada rendezvous communication between tasks whenever possible.

Here "global" means unprotected object or package on which two tasks may operate in such a way that the second task starts operation before the first has completed its operations.

(a) Wherever a global is shared between tasks, e.g., for efficiency, this fact should be clearly documented by comments.

(b) A global shared between tasks should NEVER form part of the local state of a task, e.g., the values of expressions in the task body should not depend on globals.

-- e.g., if this happens to expressions governing select alternatives, any kind of monitoring for liveness problems becomes an impossibility.

(c) The safety of overlapping operations by tasks on a shared global should NEVER depend on the underlying runtime scheduling.

*** (a) and (b) can be checked and such tools would be helpful in debugging tasking and in maintenance. At present there is not much hope of doing dynamic liveness detection for liveness errors resulting from use of shared globals, so such checkers are very very important, especially for (b) as applications requiring efficient tasking become commonplace and shared globals are used.

- .3 Try very hard not to use tasks in subprograms or local blocks; declare task objects, or allocate tasks only at outermost declarative parts of main programs or packages.
-- aimed at simplifying program structure, and alleviating the chances of liveness and termination errors.
- .4 Avoid nesting "select" and "accept" statements.
-- common sources of liveness errors.

10. Program Structure

- .1 Collect commonly used types in a global package (do not distribute them around the system)
- .2 Bounds on Tasking: Generally the total number of tasks needed for a computation should be known at compile time.
If the total number of tasks that can be initiated is not compile time determinable, this could mean that tasking is being used as a substitute for recursion; such situations should be analyzed carefully.
- .3 Tasks versus Packages: tasks provide critical sections for mutual exclusion and communication among parallel threads of control. Tasks basically are packages (with certain restrictions); in order to prevent a task becoming a global bottleneck, the programmer should attempt to make tasks "as small as possible" - this a a vague concept!
 - (a) Test One: How many other tasks may call a given service task?
If this a high proportion of the total number of tasks the programmer should check if the task can be broken into separate tasks, or if the task body can be reduced in size, perhaps by placing local data structures into a package global to the task and reducing the task to the function of scheduling access to that package.

11. Exceptions

- .1 Predefined exceptions should be handled somewhere in the program, at least in the main program.
- .2 Exceptions should NOT be used to terminate loops, nor in place of any other normal control statement.
- .3 Exception handlers should be documented by comments stating conditions under which the handler "expects" to see the exception. Similarly subprograms that may propagate exceptions should be documented with comments stating conditions under which the exception may be propagated.

-- aids in analysis of correct functioning of the program in exceptional situations.

APPENDIX B. THE PROBLEM QUESTIONNAIRE

A. I. & D. S.

Software Maintenance Questionnaire

The purpose of this questionnaire is to gather information about the software maintenance process in C3I applications.

This questionnaire is split into three parts:

REASONS FOR SOFTWARE MODIFICATIONS
SOFTWARE MAINTENANCE FUNCTIONS
DIFFICULTIES

Please base your responses on your actual experience or on the collective experience of your staff with C3I systems.

Reasons for Software Modifications

We have found that requests or reasons for software modifications can be classified into four groups. Please estimate the percentage of requests that falls into each of these categories:

- ___ modifying - New or modified requirements are defined.
- ___ adaptive - The external operating environment changes (hardware or software).
- ___ corrective - A system failure (i.e., bug) is discovered). A failure may be due to nonconformity to requirements specifications, including efficiency requirements and programming standards.
- ___ perfective - A performance enhancement is desired. These are changes that are similar to corrective changes, but are not required to make the system correct in any sense. Within existing requirements specifications, such changes may provide improvements in system capability, efficiency, maintainability, etc.

Software Maintenance Functions

The maintenance process consists of a number of functions (some of which overlap with design process functions). Please rate these functions, according to the amount of time spent doing each (during maintenance only) on a scale from 0 to 10, with 0 signifying "no time spent" and 10 signifying "extreme amounts of time spent."

- reanalysis and respecification of requirements
- redesign (high and low level)
- recoding (all code changes, including patching)
- retesting
- redocumentation
- monitoring, problem detection, diagnosis, and resolution (but not actual fixing)
- configuration control (control of code and documentation)
- training of new maintenance personnel and users
- management (formulating and following maintenance policies and procedures)

Difficulties

We have identified four particularly significant problems in the maintenance applications we have studied. Please rate these problems based on how important you feel it is to solve or eliminate them. Use a scale from 0 to 10, with 0 signifying "the problem is not worth solving" and 10 signifying "it is extremely important to solve the problem."

If there are other major problems you feel should be on this list, please add them to the bottom of this page and rate them. Also, for each major problem listed below (or added by you), please describe specific difficulties you have had (e.g., under "lack of good documentation," specific difficulties might be non-existent user documentation or source code documentation that is out of phase with the code).

- ___ difficulty in understanding what the system does because of the lack of good documentation
- ___ difficulty in monitoring operations and diagnosing problems
- ___ difficulty in deciding all relevant places to make changes (i.e., at all appropriate development levels, in all required parts of a given level, and with appropriate updating of documents and versions)
- ___ high turnover of maintenance personnel and lack of qualified replacements

Additional problems and details (use back of page if necessary):

APPENDIX C. THE CRITERIA EVALUATION QUESTIONNAIRE

A. I. & D. S.

Criteria for the Effectiveness of Software Tools

The purpose of this questionnaire is to help determine and understand the important factors in evaluating software maintenance tools. To do this, we are asking your help in assigning weights to a number of potential evaluation factors. The weights indicate how important you consider each factor, and will be used to determine overall desirability for a number of tools.

Note that at this time we are concerned only with establishing weights; we are NOT concerned with evaluating any particular tools.

Each criterion addresses a particular characteristic on which a tool can be rated. The importance of each criterion should be ranked on an integer scale from 0 to 10, with 0 signifying "this criterion is of no importance" to 10 signifying "this criterion is of extreme importance".

The evaluation criteria are divided into four major groups:

- FEASIBILITY
- CAPABILITIES AND BENEFITS
- COSTS
- QUALITIES

We realize that some of these categories are broad and overlapping, while others may not be applicable to all situations. We ask that you use your best judgment in making these evaluations.

[NOTE - The average results from this questionnaire are listed in parentheses in front of each criterion.]

FEASIBILITY

(5.6) maturity of tool technology area

This refers to the technology in general; the following criterion refers to the specific technique (e.g. parsing can be thought of as a technology area, while LR parsing is a specific technique).

Some factors that should be considered are: time in existence, number of systems developed, number of successful systems developed, and number of people involved.

(6.2) maturity of specific techniques to be used

Some factors that should be considered are:

time in existence, number of systems developed, number of successful systems developed, and number of people involved.

>>> The next four criteria concern the risk involved. For these categories, 0 signifies "high risk acceptable" and 10 signifies "no risk acceptable".

(3.8) risk in building tool

(4.1) risk in introducing tool

(7.5) risk in using tool

(8.1) tool's effect on risk in using target

Will the system be successful?

(7.4) feasibility for target language

Feasibility for some tools depends on the selected target language (though some tools are language independent). Some factors that should be considered are: existence of problem attacked by tool, and amenability of language to tool's approach.

(6.9) feasibility of tool in language's environment

Take into consideration interaction with the existing environment(s) for the language and other tools planned for the environment(s).

(6.1) fits into Air Force maintenance technical schemes

Consider the specific programming methodologies and techniques used by the Air Force (e.g. structured design).

(5.6) fits into Air Force maintenance management schemes

Consider the specific management techniques that are used by the Air Force.

CAPABILITIES AND BENEFITS

- (8.4) responsiveness to requirements
How responsive is the tool to the requirements? Consider the relative size of the problem(s) that the tool addresses and the degree to which the tool helps to solve the problem(s).
- (5.2) usefulness in other areas of life cycle
Can the tool also be used elsewhere?
- (7.7) handling frequent changes
Can the tool handle frequent and/or small changes? Does it have the ability to handle changes incrementally?
- (7.5) handling increasing complexity
Does the tool help to deal with more complex target programs?
- (6.3) management control provided by tool
Does the tool increase or aid effective management control?

COSTS

>>> For all of the cost criteria, a 0 signifies "high cost is acceptable" and a 10 signifies "low cost is required".

(3.8) time to build tool

(5.8) time to introduce tool

(7.5) time to use tool

The overall elapsed time should be considered here.

(4.3) people to build tool

(5.6) people to introduce tool

(7.2) people to use tool

Personnel considerations include:

man-months, change in productivity, change in job satisfaction, and level of user acceptance.

(5.7) software efficiency of tool

(7.3) tool's effect on software efficiency of target

Consider program size and cpu time.

(4.5) hardware costs to build tool

(6.1) hardware costs to use tool

(6.5) tool's effect on hardware costs to use target

Consider requirements of cpu speed, physical memory size, disk space, special resources, etc.

(3.9) money to build tool

(5.3) money to introduce tool

(7.5) money to use tool

(8.0) tool's effect on money required to use target

What is the overall cost (in dollars)?

QUALITIES

- (9.0) usability of tool
(8.6) tool's impact on usability of target
Consider factors such as human engineering and classes of users.
- (8.5) correctness of tool
(8.7) tool's impact on correctness of target
Can the system be formally proven correct? Or does the system inherently lack provability?
- (8.6) reliability of tool
(8.8) tool's impact on reliability of target
Consider factors such as: number of bugs found over some time period, confidence the users have in the system, and number of complaints reported.
- (7.9) testability of tool
(8.8) tool's impact on testability of target
How easy is it to test? How much confidence is there in the test results?
- (7.3) robustness of tool
(7.5) tool's impact on robustness of target
Can the system provide (help provide) automatic responses to minor problems? Can it handle user variability?
- (7.4) maintainability of tool
(9.0) tool's impact on maintainability of target
Is the system easy to understand and modify?
- (5.9) portability of tool
(6.2) tool's impact on portability of target
Can the system be moved to other environments (e.g. other machines, operating systems, target languages)?

APPENDIX D. TOOL EVALUATION QUESTIONNAIRE - AIR FORCE EVALUATORS

A. I. & D. S.

Evaluation of Software Maintenance Tools

The purpose of this questionnaire is to evaluate software maintenance tools. To do this, we are asking your help in ranking tools with respect to a set of evaluation criteria.

We have included descriptions for each of the tools to be evaluated. For each tool, we ask that you read its description and then answer the questionnaire for that tool. Each criterion in the questionnaire addresses a particular characteristic on which a tool can be rated. A description of the rating scale precedes each set of questions. The criteria in this questionnaire comprise roughly half the criteria that will be used to evaluate the tools; the remaining criteria will be evaluated by the AI&DS tool design team.

The evaluation criteria are divided into five sections:

- FEASIBILITY
- CAPABILITIES AND BENEFITS
- COSTS
- QUALITIES
- OVERALL

We realize that some of these categories are broad and overlapping, while others may not be applicable to all situations. We ask that you use your best judgment in making these evaluations.

The tool descriptions come from the draft of a report on software maintenance, being prepared by AI&DS for the Air Force. Any comments on the tools (or the descriptions themselves) are welcome.

FEASIBILITY

>>> For the following risk categories, 0 signifies "bad (large) risk" and 10 signifies "good (small) risk".

- ___ risk in introducing tool
- ___ risk in using tool
- ___ tool's effect on risk in using target software

>>> For the following categories, 0 signifies "poor fit" and 10 signifies "good fit".

- ___ tool fits into Air Force maintenance technical schemes
Consider the specific programming methodologies and techniques used by the Air Force (e.g. structured design).
- ___ tool fits into Air Force maintenance management schemes
Consider the specific management techniques that are used by the Air Force.

CAPABILITIES AND BENEFITS

>>> For the next question, 0 signifies "not responsive" and 10 signifies "highly responsive".

_____ responsiveness to requirements

How responsive is the tool to the problems it was meant to address?
Consider the relative size of the problem(s) that the tool addresses
and the degree to which the tool helps to solve the problem(s).

COSTS

>>> For the following cost criteria, 0 signifies "expensive" and 10 signifies "inexpensive".

___ time to introduce tool

___ time to use tool

The overall elapsed time should be considered here.

___ people to introduce tool

___ people to use tool

Consider factors such as required number of man-months, user acceptance, required support, etc.

___ money to introduce tool

___ money to use tool

Other costs (not including those listed above and not including hardware costs).

QUALITIES

>>> For the following criterion, 0 signifies "low satisfaction" and 10 signifies "high satisfaction".

___ usability of tool

>>> For the remaining criteria, the scale is an integer scale from -10 to +10. -10 signifies a "negative effect," 0 signifies "no effect," and +10 signifies a "positive effect."

The term "target software" refers to the software that the tool will be applied to (i.e., the application software).

___ tool's impact on usability of target software
Does the tool make the software easier to use?

___ tool's impact on reliability of target software
Does the tool help increase the reliability of the target software?
Does it decrease the number of bugs found in the software
(over some period of time)? Does it increase confidence users
have in the system?

___ tool's impact on testability of target software
Does the tool make it easier to test software? Does it increase
confidence in the test results?

___ tool's impact on robustness of target software
Can the tool help provide better handling for errors and
variability in the software environment?
Can the system provide (help provide) automatic responses to
minor problems? Can it help handle user variability?

___ tool's impact on maintainability of target software
Does the tool make the software easy to understand and modify?

___ tool's impact on portability of target
Can the tool improve or assist in moving the software to other
environments (e.g., other machines, operating systems, target
languages)?

OVERALL

>>> Finally, we would like to get some overall ratings.
For the following questions, 0 signifies "poor" and
10 signifies "excellent."

___ rate the tool on overall "practical" usefulness (i.e., how useful
could the tool be, given real-world constraints in both building and
using the tool?)

___ rate the tool on overall "theoretical" usefulness (i.e., how useful
could the tool be, given that it could be built and used effectively?)

APPENDIX E. TOOL EVALUATION QUESTIONNAIRE - AI&DS EVALUATORS

A. I. & D. S.

Evaluation of Software Maintenance Tools

FOR USE BY AI&DS EVALUATION TEAM

The purpose of this questionnaire is to evaluate software maintenance tools. To do this, we are asking your help in ranking tools with respect to a set of evaluation criteria.

We ask that you read the description for each tool, and then fill in the questionnaire for that tool. Each criterion in the questionnaire addresses a particular characteristic on which a tool can be rated. Most of the criteria are to be ranked on an integer scale from 0 to 10. The meaning of the scale is described in each section, by providing definitions of the endpoints. The criteria in this questionnaire comprise roughly half the criteria that will be used to evaluate the tools; the remaining criteria will be evaluated by outside evaluators.

The evaluation criteria are divided into five sections:

FEASIBILITY
CAPABILITIES AND BENEFITS
COSTS
QUALITIES
OVERALL

We realize that some of these categories are broad and overlapping, while others may not be applicable to all situations. We ask that you use your best judgment in making these evaluations.

FEASIBILITY

>>> For the following maturity categories, 0 signifies "very immature" and 10 signifies "very mature".

___ maturity of tool technology area

This refers to the technology in general; the following criterion refers to the specific technique (e.g. parsing can be thought of as a technology area, while LR parsing is a specific technique). Some factors that should be considered are: time in existence, number of systems developed, number of successful systems developed, and number of people involved.

___ maturity of specific techniques to be used

Some factors that should be considered are: time in existence, number of systems developed, number of successful systems developed, and number of people involved.

>>> For the following risk category, 0 signifies "bad (high) risk" and 10 signifies "good (low) risk".

___ risk in building tool

>>> For the following categories, 0 signifies "poor feasibility" and 10 signifies "good feasibility".

___ feasibility for target language

Feasibility for some tools depends on the selected target language (though some tools are language independent). Some factors that should be considered are: existence of problem attacked by tool, and amenability of language to tool's approach.

___ feasibility of tool in language's environment

Take into consideration interaction with the existing environment(s) for the language and other tools planned for the environment(s).

CAPABILITIES AND BENEFITS

>>> For the following questions, 0 signifies "low satisfaction" and 10 signifies "high satisfaction".

- ___ usefulness in other areas of life cycle
Can the tool also be used elsewhere?
- ___ handling frequent changes
Can the tool handle frequent and/or small changes? Does it have the ability to handle changes incrementally?
- ___ handling increasing complexity
Does the tool help to deal with more complex target programs?
- ___ management control provided by tool
Does the tool increase or aid effective management control?

COSTS

>>> For the following cost criteria, 0 signifies "expensive" and 10 signifies "inexpensive".

___ time to build tool

The overall elapsed time should be considered here.

___ people to build tool

Consider factors such as required number of man-months, user acceptance, required support, etc.

___ software efficiency of tool

___ tool's effect on software efficiency of target

Consider program size and cpu time.

___ hardware costs to build tool

___ hardware costs to use tool

___ tool's effect on hardware costs to use target

Consider requirements of cpu speed, physical memory size, disk space, special resources, etc.

___ money to build tool

___ tool's effect on money required to use target

Other costs (not including those listed above).

QUALITIES

>>> For the following criterion, the scale is an integer scale from -10 to +10. -10 signifies a "negative effect," 0 signifies "no effect," and +10 signifies a "positive effect."

The term "target software" refers to the software that the tool will be applied to (i.e., the application software).

___ tool's impact on correctness of target
Can the tool help prove formal correctness?

>>> For the remaining criteria, 0 signifies "low satisfaction" and 10 signifies "high satisfaction."

___ correctness of tool
Can the system be formally proven correct? Or does the system inherently lack provability?

___ reliability of tool
Consider factors such as: number of bugs found over some time period, confidence the users have in the system, and number of complaints reported.

___ testability of tool
How easy is it to test? How much confidence is there in the test results?

___ robustness of tool
Can the system provide (help provide) automatic responses to minor problems? Can it handle user variability?

___ maintainability of tool
Is the system easy to understand and modify?

___ portability of tool
Can the system be moved to other environments (e.g. other machines, operating systems, target languages)?

OVERALL

>>> Finally, we would like to get some overall ratings.
For the following questions, 0 signifies "poor" and
10 signifies "excellent".

___ rate the tool on overall "practical" usefulness (i.e., how useful
is the tool, given real-world constraints in both building and
using the tool?)

___ rate the tool on overall "theoretical" usefulness (i.e., how useful
is the tool, given that it could be built and used effectively?)

APPENDIX F. TOOL EVALUATION RESULT SCORES

This appendix contains the scores from the tool evaluation questionnaires (Appendix D and Appendix E). The scores presented here are split up by both questionnaire category (four categories: feasibility, capabilities and benefits, costs, and qualities) and by respondent (two categories: Air Force and AI&DS). Totals for each cross section are included, as well as a grand total. See section 10 for a description of the evaluation process.

Each of the scores presented is a weighted average, computed in the following way: For each criterion on the tool evaluation questionnaires, all responses were averaged, giving an average response for each criterion and tool. Then, each average response was multiplied by a weight (which reflects the importance of each criterion, as established in the previous round of questionnaires) and summed, thus resulting in a weighted average for each tool.

It is important to note that the absolute values of these scores are not as useful or important as the relative ordering of the scores. Scores between Air Force and AI&DS respondents cannot be directly compared, because Air Force respondents and AI&DS respondents had different questionnaires (and hence, different questions with different weights). Similarly, scores between different question categories cannot be meaningfully compared.

Section 10 presented the grand totals in terms of percentage of maximum value. The scores in this appendix can be converted similarly, based on the weights in Appendix C. The maximum value for any question is simply its weight multiplied by ten (since ten is the highest rating available on the second questionnaire). The maximum value for a set of questions is the sum of the maximum values for the individual questions.

FEASIBILITY

<u>Air Force</u>	<u>AI&DS</u>	<u>Total</u>	<u>Tool</u>
179.4	179.8	359.2	Programming Manager
187.9	171.3	359.2	Intelligent Editor
198.6	186.0	384.6	Documentation Assistant
199.6	193.4	393.0	Style Analyzer
200.0	203.2	403.2	Metrics Set
133.3	164.1	297.4	Annotation Language
184.0	187.3	371.3	Change Propagation Detector
165.5	174.4	339.9	Test Case Analyzer
206.2	193.2	399.4	Intelligent Tutor

CAPABILITIES AND BENEFITS

<u>Air Force</u>	<u>AI&DS</u>	<u>Total</u>	<u>Tool</u>
51.2	253.7	304.8	Programming Manager
43.5	199.8	243.3	Intelligent Editor
53.5	225.3	278.7	Documentation Assistant
48.1	154.5	202.6	Style Analyzer
51.2	139.2	190.3	Metrics Set
37.4	131.3	168.7	Annotation Language
51.2	166.4	217.6	Change Propagation Detector
46.6	145.6	192.2	Test Case Analyzer
56.5	119.3	175.8	Intelligent Tutor

COSTS

<u>Air Force</u>	<u>AI&DS</u>	<u>Total</u>	<u>Tool</u>
180.5	197.8	378.2	Programming Manager
218.6	197.9	416.4	Intelligent Editor
185.6	226.5	412.0	Documentation Assistant
213.2	209.9	423.0	Style Analyzer
240.8	266.0	506.8	Metrics Set
140.2	183.4	323.5	Annotation Language
210.2	228.2	438.7	Change Propagation Detector
180.5	219.7	400.2	Test Case Analyzer
210.9	240.0	450.9	Intelligent Tutor

QUALITIES

<u>Air Force</u>	<u>AI&DS</u>	<u>Total</u>	<u>Tool</u>
242.6	256.4	498.9	Programming Manager
167.4	250.8	418.2	Intelligent Editor
289.0	267.1	556.1	Documentation Assistant
243.9	198.1	442.0	Style Analyzer
245.1	226.9	471.9	Metrics Set
165.7	259.5	425.2	Annotation Language
231.3	233.4	464.7	Change Propagation Detector
258.0	270.3	528.2	Test Case Analyzer
192.8	246.6	439.4	Intelligent Tutor

TOTALS

<u>Air Force</u>	<u>AI&DS</u>	<u>Total</u>	<u>Tool</u>
653.6	887.5	1541.1	Programming Manager
617.4	819.7	1437.1	Intelligent Editor
726.7	904.8	1631.5	Documentation Assistant
704.8	755.8	1460.6	Style Analyzer
737.1	835.2	1572.3	Metrics Set
476.6	738.2	1214.8	Annotation Language
676.7	815.6	1492.2	Change Propagation Detector
650.6	809.9	1460.5	Test Case Analyzer
666.5	799.1	1465.6	Intelligent Tutor

REFERENCES

- [Ada 80] Reference Manual for the Ada Programming Language
Department of Defense, 1980.
- [Arsac 79] Arsac, J.
Syntactic Source to Source Transforms and Program Manipulation.
Communications of the ACM 22(1):43-54, January, 1979.
- [Ashcroft 77] Ashcroft, E. and W. Wadge.
Lucid, a Nonprocedural Language with Iteration.
Communications of the ACM 20(7):519-526, July, 1977.
- [Asirelli 79] Asirelli, P., et al.
A Flexible Environment for Program Development Based on a
Symbolic Interpreter.
In Proceedings, Fourth International Conference on Software
Engineering, pages 251-263. IEEE, 1979.
- [Baker 80] Baker, A. and S. Zweben.
A Comparison of Measures of Control Flow Complexity.
IEEE Transactions on Software Engineering SE-6(6):506-512,
November, 1980.
- [Barr 82] Barr, A. and E. Feigenbaum.
The Handbook of Artificial Intelligence.
William Kaufman, Inc., Los Altos, California, 1982.
- [Belady 71] Belady, L.A. and M.M. Lehman.
Programming System Dynamics.
Technical Report RC 3546, IBM Thomas J. Watson Research Center,
Yorktown Heights, New York, September, 1971.
- [Bergland 81] Bergland, G.D.
A Guided Tour of Program Design Methodologies.
Computer :13-37, October, 1981.
- [Bourne 78] Bourne, S.R.
The UNIX Shell.
The Bell System Technical Journal 57(6):1971-1990, 1978.
- [Boyer 75] Boyer, R.S., B. Elspas, and K.N. Levitt.
SELECT - A Formal System for Testing and Debugging Programs by
Symbolic Execution.
In Proceedings, International Conference on Reliable Software,
pages 234-245. ACM SIGPLAN, 1975.

- [Boyer 78] Boyer, R. and J.S. Moore.
A Computational Logic.
 Academic Press, New York, 1978.
- [Boyer 80] Boyer, R. and J.S. Moore.
A Verification Condition Generator for Fortran.
 Technical Report CSL-103, SRI International, 1980.
- [Brooks 75] Brooks, F.P.
The Mythical Man-Month.
 Addison-Wesley, Reading, Massachusetts, 1975.
- [Carter 82] Carter, B.
 On Choosing Identifiers.
SIGPLAN Notices 17(5):54-59, May, 1982.
- [Cheatham 79a] Cheatham, T.E., G.H. Holloway, and J.A. Townley.
 Symbolic Evaluation and the Analysis of Programs.
IEEE Transactions on Software Engineering SE-5(4):402-417, 1979.
- [Cheatham 79b] Cheatham, T.E., G.H. Holloway, and J.A. Townley.
 A System for Program Refinement.
 In Proceedings, Fourth International Conference on Software Engineering, pages 53-63. 1979.
- [Clarke 76] Clarke, L.A.
 A System to Generate Test Data and Symbolically Execute Programs.
IEEE Transactions on Software Engineering SE-2(3):215-222, 1976.
- [Davis 79] Davis, R.
 Statement on Computer Sciences.
 Subcommittee on Research and Development, Committee on Armed Services, United States House of Representatives, Washington D.C.
- [Devlin 81] Devlin, M.T.
Introducing Ada: Problems and Potential.
 unpublished paper, Air Force Satellite Control Facility, 1981.
- [Dewar 79] Dewar, R., A. Grand, S. Liu, J. Schwartz, and E. Schonberg.
 Programming by Refinement as Exemplified by the SETL Representation Sublanguage.
Transactions on Programming Languages and Systems 1(1):27-49, July, 1979.
- [Dewar ??] Dewar, R.
 The SETL Programming Language.
 NYU Courant Institute of Mathematical Sciences.

- [Donzeau-Gouge 79] Donzeau-Gouge, V., et al.
Mentor Program Manipulation System.
Technical Report, IRIA-Laboria, August, 1979.
- [Druffel 82] Druffel, L.E.
The Need for a Programming Discipline to Support the APSE: Where
Does the APSE Path Lead?
Ada Letters I(4), 1982.
- [Elshoff 82] Elshoff, J. and M. Marcotty.
Improving Computer Program Readability to Aid Modification.
Communications of the ACM 25(8):512-521, August, 1982.
- [Elspas 80] Elspas, B.
Rugged Jovial Environment.
Technical Report CSL Interim Report, SRI International, 1980.
- [Gable 80] Gable, A. and C.V. Page.
The Use of Artificial Intelligence Techniques in Computer-
Assisted Instruction: An Overview.
International Journal of Man-Machine Studies 12:259-282, 1980.
- [Gelperin 82] Gelperin, D. (ed.).
Draft, IEEE Standard for Software Test Documentation.
Standard for Software Test Documentation Task Group, IEEE
Computer Society, 1982.
- [Gerhart 80] Gerhart, S.L., et al.
An Overview of Affirm: A Specification and Verification System.
In Proceedings, pages 343-348. IFIP, 1980.
- [Glass 79] Glass, R.L.
Software Reliability Guidebook.
Prentice Hall, Englewood Cliffs, New Jersey, 1979.
- [Glass 81] Glass, R.L.
Persistent Software Errors.
IEEE Transactions on Software Engineering SE-7(2):162-168,
March, 1981.
- [Glasser 78] Glasser, A.
The Evolution of a Source Code Control System.
Software Engineering Notes 3(5):122-125, 1978.
- [Gloss-Soler 79] Gloss-Soler, S.
The DACS Glossary: A Bibliography of Software Engineering Terms.
Technical Report, Data & Analysis Center for Software, Rome Air
Development Center, October, 1979.

- [Good 79] Good, D., et al.
Principles of Proving Concurrent Programs in Gypsy.
ACM Sixth Symposium on Principles of Programming Languages ,
1979.
- [Habermann 81] Habermann, N., et al.
A Compendium of Gandalf Documentation.
Technical Report, Department of Computer Science, Carnegie-
Mellon University, April, 1981.
- [Habermann 82] Habermann, A.N., and D. Notkin.
The Gandalf Software Development Environment.
Technical Report, Department of Computer Science, Carnegie-
Mellon University, January, 1982.
- [Halstead 77] Halstead, M.
Elements of Software Science.
Elsevier North-Holland, New York, 1977.
- [Hoare 73] Hoare, C.A.R., and N. Wirth.
An Axiomatic Definition of the Programming Language PASCAL.
Acta Informatica 2:335-355, 1973.
- [Howden 77] Howden, W.E.
Symbolic Testing and the DISSECT Symbolic Evaluation System.
IEEE Transactions on Software Engineering SE-3(4):266-278, 1977.
- [Intermetrics 81] Intermetrics, Inc. and Massachusetts Computer Associates.
Ada Integrated Environment: Design Rationale.
Technical Report IR-684 (Interim Report), Intermetrics, Inc.,
March, 1981.
- [Jackson 82] Jackson, M.
A Practical Method of System Development.
Wang Institute of Graduate Studies, Prospectus on a short
course, 1982.
- [Jensen 74] Jensen, K. and N. Wirth.
PASCAL: User Manual and Report.
Springer-Verlag, Berlin, 1974.
- [Joy 80] Joy, W.
An Introduction to Display Editing with Vi.
Technical Report, Computer Science Division, University of
California, Berkeley, 1980.
- [Kafura 81] Kafura, D. and S. Henry.
Software Quality Metrics Based on Interconnectivity.
Journal of Systems and Software , 1981.

- [Kaiser 82] Kaiser, G., and A.N. Habermann.
An Environment for System Version Control.
Technical Report, Department of Computer Science, Carnegie-
Mellon University, February, 1982.
- [Kant 79] Kant, E.
Efficiency Considerations in Program Synthesis: A Knowledge-
Based Approach.
PhD thesis, Stanford University, 1979.
- [Kernighan 81] Kernighan, B.W. and J.R. Mashey.
The Unix Programming Environment.
Computer 14(4):12-24, April, 1981.
- [King 75] King, J.
A New Approach to Program Testing.
In Proceedings, International Conference on Reliable Software,
pages 228-233. ACM SIGPLAN, 1975.
- [King 81] King, J.
Program Reduction Using Symbolic Execution.
Software Engineering Notes 6(1):9-14, 1981.
- [Knuth 73] Knuth, D.
The Art of Computer Programming: Sorting and Searching.
Addison-Wesley, Reading, Massachusetts, 1973.
- [Krieg-Bruckner 80] Krieg-Bruckner, B. and D. Luckham.
ANNA: Towards a Language for Annotating Ada Programs.
In Symposium on the Ada Programming Language, pages 128-138.
ACM SIGPLAN, November, 1980.
- [Ledbetter 80] Ledbetter, L.E.
The Software Life Cycle Model: Implications for Program
Development Support Systems.
Technical Report, Schlumberger-Doll Research, May, 1980.
- [Lientz 80] Lientz, B.P. and E.B. Swanson.
Software Maintenance Management.
Addison-Wesley, Reading, Massachusetts, 1980.
- [London 78] London, R., et al.
Proof Rules for the Programming Language EUCLID.
Acta Informatica 10:1-26, 1978.
- [Loveman 77] Loveman, D.
Program Improvement by Source to Source Transformation.
Journal of the ACM 24(1):121-145, January, 1977.

- [Low 74] Low, J.
Automatic Coding: Choice of Data Structures.
PhD thesis, Stanford University, 1974.
- [McCune 79] McCune, B.P.
Building Program Models Incrementally from Informal Descriptions.
PhD thesis, Stanford University, 1979.
- [McCune 82] McCune, B., J. Dean, and D. Shapiro.
Rule-Based Information Retrieval.
Technical Report 1018-2, Advanced Information & Decision Systems, Mountain View, CA, April, 1982.
- [McIlroy 78] McIlroy, M., E. Pinson, and B. Tague.
UNIX Time-Sharing System: Foreword.
Bell System Technical Journal 57(6):1899-1904, 1978.
- [Medina-Mora 82] Medina-Mora, R.
Syntax-Directed Editing: Towards Integrated Programming Environments.
PhD thesis, Carnegie-Mellon University, March, 1982.
- [Melliars-Smith 82] Melliars-Smith, P.M. and R. Schwartz.
The Proof of SIFT (Software Implemented Fault Tolerance).
Software Engineering Notes 7(1):2-5, 1982.
- [Nelson 82] Nelson, T.
A New Home for the Mind.
Datamation, March, 1982.
- [Osterweil 81] Osterweil, L.
Software Environment Research: Directions for the Next Five Years.
Computer :35-43, April, 1981.
- [Perlis 81] Perlis, A.J., F.G. Sayward, and M. Shaw.
Software Metrics.
MIT Press, Cambridge, Massachusetts, 1981.
- [Polak 80] Polak, W.
Theory of Compiler Specification and Verification.
Technical Report STAN-CS-80-802, Computer Science Department, Stanford University, 1980.
- [Reifer 82] Reifer, D.J.
Increasing Software Productivity.
DPMA Seminar Description, 1982.

- [Rich 78] Rich, C. and H. Shrobe.
Initial Report on a Lisp Programmer's Apprentice.
IEEE Transactions on Software Engineering SE-4(6):456-467,
November, 1978.
- [Rich 81] Rich, C.
Inspection Methods in Programming.
PhD thesis, MIT, June, 1981.
- [Robinson 79] Robinson, L.
The HDM Handbook.
Technical Report Vols. I - III, SRI International, 1979.
- [Sale 81] Sale, A.H.J.
Proposal for Extension to Pascal: Addition of REPEAT and UNTIL
as Identifiers.
ACM SIGPLAN Notices 16(4), April, 1981.
- [Schwartz 75] Schwartz, J.
Automatic Data Structure Choice in a Language of Very High
Level.
Communications of the ACM 18(12):722-728, December, 1975.
- [Shapiro 81] Shapiro, D.
Sniffer: A System that Understands Bugs.
Technical Report MIT/AIM/638, MIT AI Lab, June, 1981.
- [Shapiro 82] Shapiro, D., B. McCune, and G. Wilson.
Design of an Intelligent Program Editor.
Technical Report 3023-1, Advanced Information & Decision
Systems, September, 1982.
- [Stoneman 80] Requirements for Ada Programming Support Environments.
Department of Defense, 1980.
- [SVG 79] Stanford Verification Group.
Stanford Pascal Verifier User Manual.
Technical Report STAN-CS-79-731, Computer Science Department,
Stanford University, 1979.
- [Teitelbaum 80] Teitelbaum, T. and T. Reps.
The Cornell Program Synthesizer: A Syntax Directed Programming
Environment.
Technical Report TR 80-421, Department of Computer Science,
Cornell University, May, 1980.
- [Teitelman 78] Teitelman, W.
Interlisp Reference Manual
Xerox Palo Alto Research Center, 1978.

- [Teitelman 81] Teitelman, W. and L. Masinter.
The Interlisp Programming Environment.
Computer 14(4):25-33, April, 1981.
- [Tichy 80] Tichy, W.
Software Development Control Based on System Structure Description.
PhD thesis, Carnegie-Mellon University, January, 1980.
- [TRW 80] A Study of Embedded Computer Systems Support.
TRW (performed for the Air Force Logistics Command, Wright Patterson AFB), 1980.
- [Unix 80] Unix User's Manual
Bell Laboratories, Murray Hill, New Jersey, June 1980.
- [Waterman 78] Waterman, D.
Rule-Directed Interactive Transaction Agents: An Approach to Knowledge Acquisition.
Technical Report R-2171-ARPA, Rand Corporation, February, 1978.
- [Waters 82] Waters, R.
The Programming Apprentice: Knowledge Based Program Editing.
IEEE Transactions on Software Engineering SE-8(1), January, 1982.
- [Wescourt 77] Wescourt, K., M. Beard, and L. Gould.
Knowledge-Based Adaptive Curriculum Sequencing for CAI: Application of a Network Representation.
Technical Report 288, Institute for Mathematical Studies in the Social Sciences, Stanford University, September, 1977.
- [Yau 80] Yau, S.S. and J.S. Collofello.
Some Stability Measures for Software Maintenance.
IEEE Transactions on Software Engineering SE-6(6):545-552, November, 1980.
- [Zelkowitz 78] Zelkowitz, M.
Perspectives on Software Engineering.
Computing Surveys 10(2):197-216, June, 1978.



*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.

4-8
DTI