83N20591

Final Report

on NASA Grant No. NAG-1-233

ON THE ENGINEERING OF CRUCIAL SOFTWARE

Submitted to:

National Aeronautics and Space Administration
Langley Research Center
Hampton, Virginia 23665

Attention: A. O. Lupton
Mail Stop 477

Submitted by:

Terrence W. Pratt
Professor

John C. Knight
Associate Professor

Samuel T. Gregory

Report No. UVA/528208/AMCS83/102

February 1983

DEPARTMENT OF APPLIED MATHEMATICS
AND COMPUTER SCIENCE

Final Report

on NASA Grant No. NAG-1-233

ON THE ENGINEERING OF CRUCIAL SOFTWARE

Submitted to:

National Aeronautics and Space Administration
Langley Research Center
Hampton, Virginia 23665

Attention: A. O. Lupton
Mail Stop 477

Submitted by:

Terrence W. Pratt
Professor

John C. Knight
Associate Professor

Samuel T. Gregory

Department of Applied Mathematics and Computer Science

RESEARCH LABORATORIES FOR THE ENGINEERING SCIENCES

SCHOOL OF ENGINEERING AND APPLIED SCIENCE

UNIVERSITY OF VIRGINIA

CHARLOTTESVILLE, VIRGINIA

Report No. UVA/528208/AMCS83/102

Copy No. _______

February 1983

1. Report No.	2. Government Accession No.	3. Recipient's Catalog No.
4. Title and Subtitle On the Engineering of Crucial Software		5. Report Date February 1983 6. Performing Organization Code 5-28208
7. Author(s) John C. Knight and Samuel T. Gregory		8. Performing Organization Report No. UVA/528208/AMCS83/102 10. Work Unit No.
9. Performing Organization Name and Address University of Virginia Thornton Hall Charlottesville, VA 22901		11. Contract or Grant No. NAG-1-233 13. Type of Report and Period Covered Final Report 11/01/81 - 10/31/82
12. Sponsoring Agency Name and Address National Aeronautics and Space Administration Langley Research Center Hampton, Virginia 23665		14. Sponsoring Agency Code

15. Supplementary Notes

None

16. Abstract

This report discusses the issues involved in building software for crucial applications. These are applications in which failure could endanger expensive equipment or threaten human life.

The conventional software development cycle is discussed and various enhancements suggested based on recent results in software engineering. It is argued that the conventional software development cycle is inadequate for crucial applications even if enhanced.

An alternative approach is proposed in which human creativity is removed from software development as far as possible and replaced by computer based program synthesis. This technology is relatively immature but offers great potential for improving reliability of software. For those parts of systems which cannot presently use this approach because the technology is inadequate, fault tolerance is proposed as a supplement to the conventional software cycle.

This work was undertaken to provide suggestions to the sponsor about promising research areas. Specific research suggestions are made as well as suggestions for experiments using the AIRLAB facility. The report contains extensive bibliographies on various related topics to provide sources of further reading for those areas not covered in sufficient depth in the report.

17. Key Words (Suggested by Author(s))	18. Distribution Statement
software reliability, reliable software development, fault tolerance, automatic programming	Unclassified

19. Security Classif. (of this report)	20. Security Classif. (of this page)	21. No. of Pages	22. Price
Unclassified	Unclassified	230	

For sale by the National Technical Information Service, Springfield, Virginia 22161

ABSTRACT

The most significant shortcoming of all software development processes lies in the fact that humans are involved.

TABLE OF CONTENTS

Table of Contents ii

List of Figures vi

1 Introduction 1

2 THE SOFTWARE DEVELOPMENT CYCLE 3

2.1 Requirements Specifications 5

2.1.1 State of the Art 5

2.1.2 Contribution to Reliability 6

2.1.3 Activity Centers 8

2.2 Design Methodology 11

2.2.1 Contribution to Reliability 11

2.2.2 State of the Art 12

2.3 Programming Languages 16

2.3.1 Introduction 16

2.3.2 Modula-2 16

2.3.3 HAL/S 19

2.3.4 Ada 23

2.3.5 Summary 27

2.4 Testing 30

2.4.1 State of the Art 30

2.4.2 Contribution to Reliability 32

2.4.3 Testing Techniques 33

2.5 Programming Environments 41

3 Enhancements To The Conventional Software Development Cycle
.... 50

3.1 Overview 50

3.2 Software Prototypes 52

3.3 Software Components 55

3.4 Integrated Environments 58

3.5 An Improved Conventional Software Development Cycle 60

4 The Inadequacy Of The Software Development Cycle 65

5 Fault Tolerance 68

5.1 Recovery Blocks 69

5.2 N-Version Programming 71

5.3 Reliability Improvement 73

6 Verification 74

7 Automatic Programming 76

7.1 Introduction 76

7.2 The Issues in Automatic Programming 81

7.3 Automatic Programming Systems 84

7.4 Conclusions About Automatic Programming 87

8 A Comprehensive Approach 88

8.1 Overview 88

8.2 Requirements 91

8.3 The Monitor 92

8.4 The Automatic Programming System 93

8.5 Verification 95

8.6 Conventional Software Development Cycle 96

8.7 Fault Tolerance 97

8.8 Testing 98

9 AIRLAB Research and Experimentation Recommendations 101

9.1 The Software Development Cycle 102

9.2 Fault Tolerance 104

9.3 Automatic Programming 106

9.4 Comprehensive Approach 108

10 Conclusions 109

REFERENCES 112

INTRODUCTION TO THE BIBLIOGRAPHIES 125

BIBLIOGRAPHY ON REQUIREMENTS ENGINEERING 127

BIBLIOGRAPHY ON DISTRIBUTED SPECIFICATIONS 138

BIBLIOGRAPHY ON DESIGN METHODOLOGIES 141

BIBLIOGRAPHY ON PARALLEL PROGRAMMING LANGUAGES 153

BIBLIOGRAPHY ON TESTING METHODOLOGIES 160

BIBLIOGRAPHY ON STATIC ANALYSIS 173

BIBLIOGRAPHY ON SYMBOLIC EXECUTION 175

BIBLIOGRAPHY ON PROGRAMMING ENVIRONMENTS 176

BIBLIOGRAPHY ON SOFTWARE PROTOTYPING 180

BIBLIOGRAPHY ON FUNCTIONAL LANGUAGES 181

BIBLIOGRAPHY ON VERIFICATION 184

BIBLIOGRAPHY ON FAULT TOLERANCE 210

BIBLIOGRAPHY ON AUTOMATIC PROGRAMMING 219

LIST OF FIGURES

Figure 2.1 4

Figure 3.1 61

Figure 8.1 90

Figure 8.2 99

ACKNOWLEDGEMENTS

UNIX is a trademark of Bell Laboratories,

Ada is a trademark of the U.S. DoD.

and NonStop is a trademark of Tandem Computers.

It is a pleasure to acknowledge financial support for this work received under NASA grant number NAG-1-233.

SECTION 1

Introduction

Crucial software is any software whose failure could endanger human lives or threaten the safety of expensive equipment. For example, the software in computers providing active controls for aircraft is crucial.

Software is defined to be reliable if it complies with its requirements specification most of the time. Conversely, software is said to have failed when it no longer complies with its requirements specification. We choose not to define 'most' because that leads to an attempt to quantify software reliability and the goals of this grant do not include probabilistic and statistical analysis of software failures. Rather, we assume that any increase in reliability is desirable and any methodology which may bring about an increase is worthy of consideration. We assume that the determination of whether an increase has been achieved is ascertained by experiments using conventional statistical methods.

The purpose of this grant was to examine and extend a preliminary approach to the engineering of crucial software which was presented in the original grant proposal. The goals were to prepare a comprehensive approach together with recommendations of those areas of software technology which are most likely to produce a substantial improvement in software quality if vigorously pursued. Our primary conclusion from extensive reviews of the literature and discussions with numerous

experts is that it is inappropriate at this time to propose a single comprehensive approach to crucial software development. Rather, we find several complementary technology areas which seem to offer the potential of major increase in software reliability yet which are not sufficiently mature that a clear choice can be made as to which is most appropriate.

This report is divided into ten sections. In Section 2, we examine the various aspects of the conventional software development cycle. This cycle was the basis of the augmented approach contained in the original grant proposal. We have formed the opinion that this cycle is inadequate for crucial software development, and the justification for this opinion is presented in Section 3. In Section 4 several possible enhancements to the conventional software cycle are discussed. Software fault tolerance is a possible enhancement of major importance and is discussed separately, in depth, in Section 5. Formal verification using mathematical proof is considered briefly in Section 6. Automatic programming is a radical alternative to the conventional cycle and is discussed in Section 7. Our recommendations for a comprehensive approach are presented in Section 8, and various experiments which could be conducted in AIRLAB are described in Section 9. Our conclusions are presented in Section 10. Finally, we present extended bibliographies on the topics covered in this report. They are intended to provide the reader with starting points for exploring further any of the subjects addressed in this report.

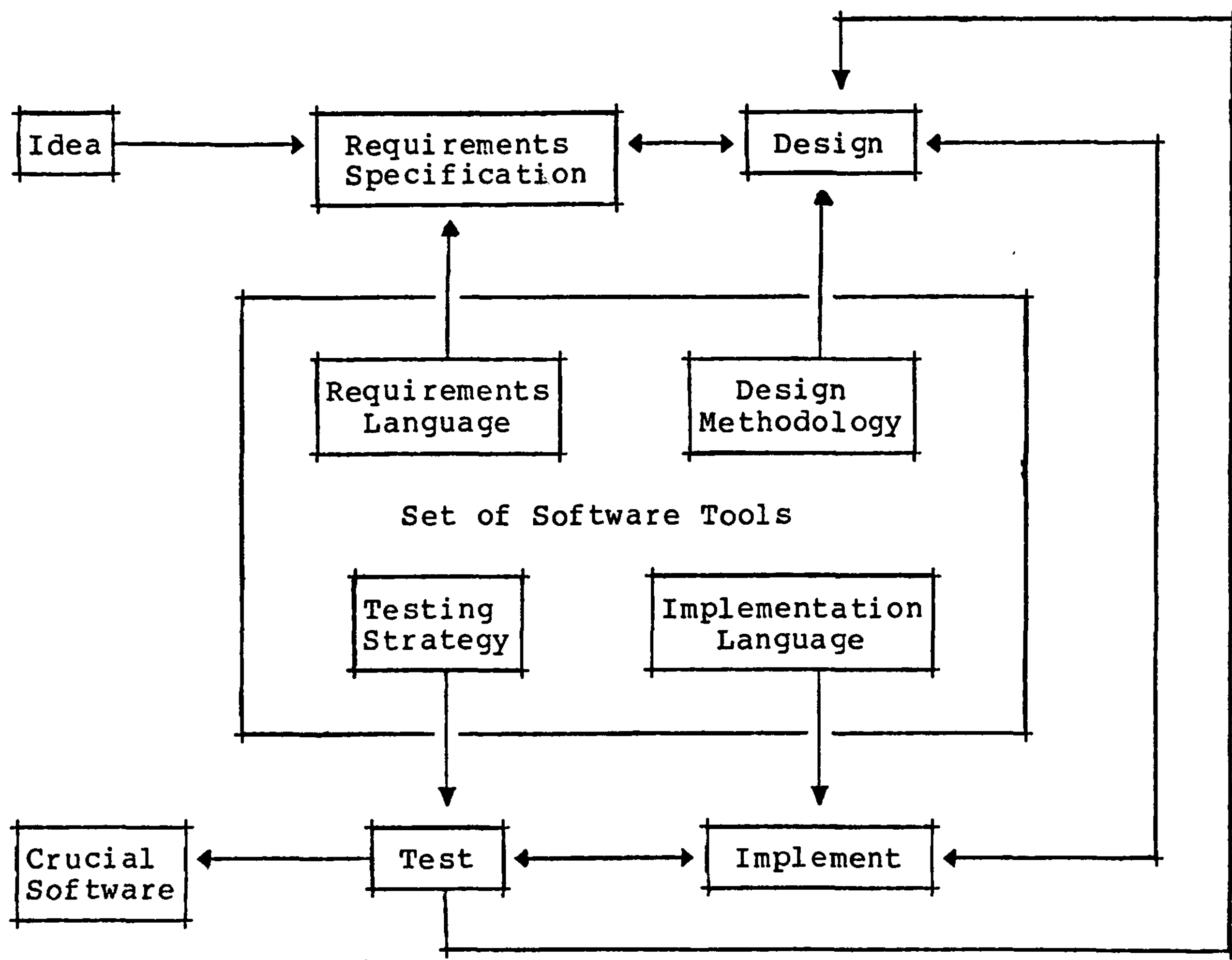
SECTION 2

THE SOFTWARE DEVELOPMENT CYCLE

In the short term, the only feasible way to construct crucial software is to use all of the best available tools and technologies, and to apply them in the classical software development cycle. Even then, they may not yield the required quality, but this determination is specific to the system and the people involved in its creation.

The software development cycle which we are discussing in this section is shown in Figure 2.1. It consists of only those steps typically used at the present time in the development of software systems. As such, it is a starting point for discussion and is simpler than the approach contained in the original proposal for this grant.

In our review of the present state of the art, we have formed certain conclusions which relate to elements of the classical software development cycle, each of which is discussed briefly in the following sections.



Conventional Software Development Cycle

Figure 2.1

2.1. Requirements Specifications

A requirements specification is a formally written statement of what a software system is supposed to do rather than (as in a design document or the actual code) how the system is to do it. Here, what is required of the system is explicitly written down and can be reviewed with the customer at the earliest stages to verify that the system to be built actually reflects what is wanted. The creation of such a document affords an early opportunity to review the consistency and completeness of the idea so problems can be corrected before their consequences proliferate. If the requirements specification is written in a formal requirements language, it is possible to perform some consistency checks automatically.

2.1.1. State of the Art

Many projects, such as the original development of the A-7E software [1], surge ahead into design without ever finding out what the software system is supposed to do. Others do attempt to organize the requirements specification in English prose, producing large documents in which it is easy to get lost, which are often incomplete or wrong (e.g. not specifying functions which the customer wants), and which are often never read nor kept up to date (this was the case with the original 2500 page BMD [2] requirements specification).

There is a great deal of current activity in the development of requirements languages and analyzers. Some of the older attempts are merely text organizers which are incapable of much more than cross

referencing usages of words in the document [3]. An apparently successful method [1] provides suggestions of how to design and use forms to be filled out about the project rather than a language per se. There are those who contend that requirements languages should be especially designed for restricted application areas, thus we find people working on requirements language generators [4]. Work has also been done towards developing programs which will automatically perform consistency and completeness checks on machine-readable requirements specifications [5].

It seems to be somewhat easier to write down the requirements for business systems and for purely mathematical software than for real-time systems. As a consequence, languages for those areas are much more advanced. In the area of real-time software, hardware interfaces and timing limitations must be specified, and priorities of goals must be stated in anticipation of necessary optimizations. Just how best to express a requirements specification continues to be an area of investigation.

2.1.2. Contribution to Reliability

By definition, reliability of a software system involves the degree of its fidelity to its requirements specification. The requirements specification should be written before further work on the system is started. The requirements specification can be used to verify with the customer that the developers understand exactly what is required, and then can be used as a reference by the developers in making all deci-

sions regarding the project. The continual reference to an explicit statement of what the product is to do cannot but help to ensure the product's fidelity to those requirements.

If at all possible, the document should be written in a requirements language. When requirements analyzers become available, this would allow automated completeness and consistency checks, this is especially important for changes during the post-delivery phase of the software system's life. Requirements languages are designed to avoid some of the problems of natural languages. Part of the power of the English language lies in its ambiguity and the extensive use of context to convey meanings. Ambiguity is inherently unsafe. For example, although not a requirements specification, the Ada Reference Manual [6] has for several years been a source of controversy over the meaning of the language it is supposed to describe. Further, in a natural language it is too easy to omit parts of the requirements specification, and the very structure of the language prevents explicit connection of a network of interrelationships.

A statement of requirements serves as a reference of what the software system is really supposed to do, thus it serves as a "contract" with the customer and with the eventual user, and can guide decisions during design and coding. This helps to prevent "guesses" by design analysts and programmers. It is far easier to detect and remove basic concept faults before design than after much of a software system has been designed (& coded!) to depend on them. A requirements specification which is organized by the use of a requirements language can be analyzed for such faults before a design exists to be infected by them.

A requirements specification helps in the creation of tests which will actually "verify" the software product since it is explicitly stated what the product is to do in every situation. Thus, each test can actually contribute to knowledge about the system's reliability, and none need be superfluous. If the software's response to an input is unspecified, _whatever response_ it gives is as valid as another. Problems of cross-accusations of what should have been assumed by whom could be avoided if _completeness_ checks are performed on a requirements specification. In this context we completely reject the notion of "robustness" in software [7]. Robust software is supposed to act "sensibly" when it receives unexpected input in the event that nothing was in the requirements specification. During the post-delivery phase of a system's life, the document continues to serve as a reference to guide proper or permitted revisions. Moreover, it is an excellent place to document the whys and wherefores of the changes, and the altered set of requirements can be checked for consistency and completeness as before. Actually, since the system would have been built around the requirements specification, any changes during this period should be due to changes in what is required of the system, which makes it appropriate to amend the requirements specification.

2.1.3. Activity Centers

For a survey of work in this area, see the May 1982 issue of IEEE Computer Magazine, in particular the chart by R.J.Lauber comparing 11 requirements languages and analysis systems on page 40 [8].

Parnas and Heninger, for the Naval Research Laboratory, Washington, D.C. while at UNC Chapel Hill, developed a requirements specification methodology which is applicable to flight software, since their project was to build a system duplicating the functionality and time and space efficiency of the A-7E aircraft operational flight program using modern software engineering techniques [1,9]. There had been no previous requirements statement for the A-7E and the document resulting from this project _is being used_ by the "maintenance staff" for the _original_ software. It is unclear how much of their success was due to the fact that they were writing requirements specifications for an existing system [1].

PDL [3] is a text organizing method with limited cross referencing capabilities and, although intended for design documents, has been used for requirements specification. A problem is that garbage text is perfectly acceptable to its processor.

PSL/PSA (Problem Statement Language/Analyzer) [5] is an older system which seems to have had some success as we find many projects have used it and there have been favorable comments about it in the literature (see the Bibliographies).

SREM (RSL/REVS) (Software Requirements Engineering Methodology) [10,11,12,13] is available from the Ballistic Missile Defense center in Huntsville. This system has actually been used in specifying the requirements of a large real-time project.

As can be seen, an explicit requirements specification is highly desirable in an effort to produce reliable software. However, the technologies of languages for its expression and analyzers of its

consistency and completeness are not yet well established. Further, there is nothing to assure that the document is actually used or kept up-to-date as the life of a project progresses.

2.2. Design Methodology

Software Design methods are largely disciplined ways of thinking through the problems the software is to solve. What the design stage is to accomplish is the translation of the "what" description of the requirements specification (most of the methodologies assume the existence of a requirements specification) into an overall plan for implementation --an overview of "how". This plan is to be written in what has come to be known as a design language; a specialized notation for accurately communicating what is to be accomplished to the individual programmers who will be implementing the system. Most of the work in discovering design methods occurred in the early to mid Seventies under the umbrella term "Software Engineering." Often, the process is seen as a continuum with only a vague distinction between "gross design" (which we are calling design) and "detailed design" (which we are calling implementation); in such cases, the design language is effectively the implementation language.

2.2.1. Contribution to Reliability

There are several motivations for preparing a design:

a) A thorough examination of the requirements specification for an implementation strategy affords the opportunity of ascertaining whether the project can be accomplished at all.

b) This same thorough coverage allows the design team to determine the most vital areas for allocation of implementing personnel. It also

allows the establishment of milestones for the development process.

c) A large system which is to perform a wide variety of functions needs a great deal of organization and planning. Creation of a design forces a disciplined approach to a problem and the resulting document serves as a guide at every stage of development. A document aimed at directing the implementors by limiting their scope of concerns can serve testers as well, indicating which areas of the software are intended to correspond to parts of the requirements specification.

d) A design document provides a mapping from the requirements specification to the coded software to limit the search for the modules affected by later revisions to the requirements.

e) This documentary evidence can be used early on as a check point for compliance with the requirements specification.

Unfortunately, the entire design process also provides more opportunities for faults to be introduced; hence, the attempts at devising analysis tools for design languages [5]. The factor which makes a design worthwhile is that faulty decisions _may_ be detected as they are made rather than later when too much work depending upon them is at stake to do more than patch.

2.2.2. State of the Art

This section provides some warnings about those methods considered more likely than others to aid in creating effective designs. None of

them is a panacea. Indeed, it has been observed that most of these methods are those which have been unconsciously employed by the best programmers for years [14,15].

Most techniques are still at a stage in which they require lots of "magic" [16] and often are described in very vague terms by their devisers (see practically anything in the Design bibliography). Two people using the same method on the same problem (requirements specification) will rarely come up with the same design (this, the result of experiment [16]). Thus software design is still a game of skill, and quite prone to human error.

The Jackson methodology [17] views a program as a transformer of the structure of its input data to that of its output. Its area of application has traditionally been in business data processing; otherwise, it has not been applied in practice to large projects. Whether the complexity of resolving structural conflicts can remain manageable has not been determined. This is representative of the "data driven" design methods.

In Dijkstra's Programming Calculus [18], the Floyd/Hoare [19,20] axioms (augmented with later developments [21]) are used to formally derive a program from its requirements specification rather than to prove an existing program. This method is not necessarily a separate step from coding, and has been found difficult for the "average" programmer to understand. This method works, in the context of algorithms involving only integers and logicals, and is included within the basis for the recommendations below, but it can easily be mis-used through inattention to strict logical detail, a failing for which humans are

notorious. The Stepwise Refinement [14] strategy (also known informally as "structured programming", "structured design", "top-down programming", and "top-down design") is often incidentally employed to limit complexity.

The trend towards including data abstraction mechanisms in programming languages reveals a renewed respect for Parnas' Information Hiding [22]. This method has also been widely misinterpreted [23]. Other terms informally used concerning methods in this category are "functional decomposition", "modularization", and "object oriented design" [24].

The Data Flow design methods [25,26,27] attack a project by analyzing the necessary itinerary of various items of information through a network of transformations which gradually evolve the outputs from the inputs. The choice of division among nodes of the network, however, can often change application of this method into application of Functional Decomposition.

Iterative Enhancement [28,29] is a simultaneous design and implementation method in which a small portion of a system's functionality is carried through to completion. This program is then given more functions piecemeal in the same manner as the original chunk. Occasionally, the original part may have been the prototype model.

SRI's Hierarchical Design Methodology [30] provides a set of tools and languages which together allow the consistent use of a combination of the above methods: top-down or hierarchical partitioning of the system (requirements specification, design, and implementation) into multi-level abstract machines, separation of the functions provided by

each machine at its level, and verification of the consistency of requirements specification, of design with requirements specification, and of the implementing code. The code-level formal proof is (or was, until recently) based upon the Boyer-Moore theorem prover [31].

The state of many design languages is evidenced by the fact that Basili's "system" is simply a means of rapidly changing the syntax of his "generic" design language [4].

A recommended overview of the more viable categories of methods, with examples, appears in [16].

A good design is vital to reliable software, but the technology for assuring production of or adherence to good designs is not there. We have not progressed far beyond explicit statement of what good programmers have always done unconsciously. The technology of design languages and analyzers is not very far advanced, nor is there any way of preventing their misuse. The apparently contained system of the HDM still only allows consistent use of design methods. There is little to require appropriate application of the system.

2.3. Programming Languages

2.3.1. Introduction

Crucial systems usually operate in real time. Modula-2, HAL/S, and Ada are high-level languages intended for real-time programming. In this section, we examine some of the facilities in each of these languages which have met with appreciation from real-time programmers and those which have been found unsatisfactory. This examination reviews the state of the art in programming languages.

2.3.2. Modula-2

Wirth claims that ordinary parallel languages contain all that is needed in a real-time language [32]. He proposes a discipline for their use in which a correct program is built first and then optimized to timing constraints. All time dependencies are confined to interrupt handlers and the program should not depend on any particular strategy for process scheduling. There has been some disagreement about this practiced ignorance of scheduling, and Wirth's second try at designing Modula, which produced Modula-2, forces the user to design his own scheduling algorithm. Confining all time dependencies to interrupt handlers cannot be done other than in programs which merely monitor devices. A program's computational processes can produce correct results, but if those results are not available for output when needed, the program is useless as a real-time program. Wirth also suggests avoiding many timing problems by adding more processors. This is fine if we have the money and space for the extra processors and necessary wiring.

However, the suggestion ignores the added problems and overhead of inter-processor communication. One suggestion that seems to get agreement from real-time programmers is that compilers should tell how long each statement and overhead operation will actually take.

Holden and Wand have programmed a loosely constrained real-time application (an operating system) in Modula [33]. They label as good the ability to give an absolute address to a variable at its declaration and complain about the difficulty of writing disk drivers without some generic parameter type. The latter problem is fixed in Modula-2 with the types WORD and ADDRESS which match almost anything. Wirth [34] claims a variable address declaration is extraneous with these "magic" types, but was included in Modula-2 at his colleagues' insistence. Holden and Wand point out that Modula's design calls for a uniform hardware I/O scheme of memory addressable "device registers" and may have problems on a different architecture such as ports with special I/O instructions. Modula-2 does assign static priorities to processes and procedures and these priorities are defined to be associated with those of interrupts in the hardware, but a dynamic priority effect can be achieved since procedures have the option of always being executed at their own declared priority rather than inheriting the priority of the process executing them. Thus, Modula-2 allows the user to determine whether his application will have priorities assigned statically or dynamically. Also, of these three languages, only Modula-2 defines what process priorities mean in relation to the environment they must deal with in real time.

Holden and Wand say that Modula's design limits its range of applications since all processes must cooperate in sharing the processor. In Modula (which Wirth [34] considers only a preliminary design for Modula-2 in the sense of Preliminary Ada and Ada), a process must consent to sending a signal and in Modula-2 a process must execute procedure TRANSFER before a process swap can take place. Complaints about lack of pre-emptability in Modula-2 seem suspect in light of the fact that pre-emption is generally achieved via interrupts as it is in Modula-2. These complaints seem to ignore the fact that Modula-2 is intended to be used in implementing facilities such as pre-emptive schedulers.

Modula [33] was a basis for the YELLOW candidate in DoD's search for a real-time language, i.e. it was a candidate design for Ada. It was found lacking in that it does not have a fixed point/floating point option, it does not provide for machine code inserts in the high-level language code, it has no exception handling capabilities, it has no facilities for specifying the machine representation of data objects, and cannot express, in one program, operation of a multiprocessor system.

Certain facts tend to cast doubts on the inherent efficiency of Modula-2 [34]. Wirth designed his Lilith machine especially for the language. Lilith is microcoded so that the instruction set is the Modula-2 specific M-code. Also, his most time-critical device, the high-resolution display, has its own bus to memory and that bus has four times the bandwidth of the CPU's bus.

Other obvious concerns with Modula-2 as a language for crucial systems are the relatively low level of typing in the language and the lack of a systematic approach to constraints. Much of this was corrected in Yellow, but that language was abandoned. Modula-2 is certainly better than assembly languages but was not designed for, and does not aid the development of crucial systems.

2.3.3. HAL/S

HAL/S [35] was adopted as a NASA standard flight language when an implementation was demonstrated to have a ten to fifteen percent inefficiency in size and speed over assembly language. We point out that this is a ridiculous metric. Efficiency is program dependent and compiler dependent. The most important issue is reliability and that is ignored.

The language itself puts the periodicity of process scheduling, control via wall clock time, events (hardware interrupts), and error conditions under explicit programmer control. These things are achieved via a large run-time library support system, and the HALMAT intermediate language operators for many of these facilities are mnemonic for IBM OS/360 supervisor calls. In contrast to Modula-2, HAL/S does not provide basic, low-level, facilities for tailoring an entire system to an application but tries to assume the class of real-time programs known as flight software and to provide a full underpinning for the user to build on. Where the user needs access to the hardware, the language provides the SUBBIT operator for bit manipulation and an implementation provides %MACRO's rather than allow assembly code insertion. This allows com-

piler checks on usage while providing high-level access to machine idiosyncrasies.

From the literature, HAL/S does not seem well known outside of INTERMETRICS and NASA. A brief description of some of its real-time related constructs follows. The words in upper case are keywords of the HAL/S language.

Outside of implementation-specific %MACRO's, there is no absolute addressing. Data storage may be AUTOMATIC (allocated only as long as a procedure is activated), STATIC (allocated as long as the program executes), or TEMPORARY (allocated only while a few statements execute). Data may be DENSE (packed), ALIGNED on unspecified "appropriate" hardware boundaries, or RIGID (laid out in memory exactly as described in the declaration). ACCESS rights may be associated with data objects and they may be grouped into LOCK groups for mutually exclusive access through UPDATE blocks by tasks. Events are boolean-like variables which may be LATCHED or not (able to hold a true value for more than an instant or not). All communication among tasks is through shared variables. Separately compiled entities access data via a FORTRAN COMMON-like facility known as COMPOOL's. Procedures and functions may be expanded INLINE or may be specified to be REENTRANT or not. A degree of optimization for common flight software applications is achieved by virtue of the special VECTOR and MATRIX operators and data types. A task may be stopped by another task by two methods: CANCEL allows the current instance of the task to continue to completion but prohibits any scheduled future instances of it, whereas TERMINATE destroys the current instance as well. A task may WAIT UNTIL a certain wall clock time, WAIT

for a certain length of time, WAIT FOR a combination of events to become true, or WAIT FOR DEPENDENT's to terminate. A hardware interrupt or a task may SET an event variable true or RESET it false or may make it true momentarily via SIGNAL. When an event variable changes values, every event expression which has been reached by any task must be fully re-evaluated to determine if the task is eligible to proceed. Error conditions within a class or entire classes of errors may be raised (SEND ERROR...), and the set of error handlers may be dynamically changed by declaring and removing them (ON ERROR... statement; and OFF ERROR...). As a special case, errors may be ignored or passed to the support system with an optional change to an event variable (ON ERROR... SYSTEM... or ON ERROR... IGNORE...).

The most attractive statement in HAL/S for the real-time programmer is the SCHEDULE statement. A task may be scheduled to begin execution AT a certain time, within (IN) a certain time interval of the current time, or ON the occurrence of true evaluation of an event expression. It is required to be started with a priority, and may be made DEPENDENT on the continued existence of the task executing the SCHEDULE statement. Execution of the task may be made to begin anew EVERY so often or a certain amount of time AFTER it completes. Such repetition may continue WHILE an event expression holds true or UNTIL an event expression becomes true or UNTIL a certain time. All this may be specified in a single SCHEDULE statement, and once started, a task's priority may be changed by the UPDATE PRIORITY statement.

On the surface, HAL/S seems to provide everything a real-time programmer could want; particularly if a compiler could guarantee the

scheduling requested in each SCHEDULE statement. Garman [36], however, describes several problems with HAL/S in the Space Shuttle project.

On occasion the project was forced to take risks by changing shared variables outside of UPDATE blocks. This casts some doubt on the utility of any language which prohibits shared variables or their unprotected update. Either the implementation (one of three [37]) of HAL/S used by the project did not support or the project did not use the following features: DEPENDENT, REPEAT AFTER, TERMINATE, WAIT UNTIL, and ON ERROR. UPDATE PRIORITY was rarely used, which implies that the need to change a process' priority is rare in real-time programs but probably vital when it does arise. Also, the implementation imposed severe limits on the complexity of event expressions that could be used. This last rule was probably imposed to cut down on overhead since all event expressions must be re-evaluated on any event change.

The original coding of the Shuttle software [36] turned out to be plagued with throughput problems. For example, the I/O via READ/WRITE statements or %MACRO's was too expensive. The project called for the various machines to synchronize at most support routine calls. And there were too many processes, resulting in scheduler queue overflows. It was also apparent that, even with a SCHEDULE statement, timing constraint calculations had to be made by hand or with the aid of FSIM, a functional simulation tool. The solution chosen was to break up certain tasks into procedures and change the support executive to call these procedures in an order determined by table-lookup, a technique employed in many assembly language real-time programs [38].

The generalized scheduling constructs of HAL/S, a language designed for flight software, were found to be too inefficient in practice and some parts were not implementable. Tripathi, Young, Good, and Brown, in describing a verifiable subset of HAL/S and before completion of a verifyability study of Ada, concluded that a project should choose Ada over HAL/S, noting that Ada has all the capability of HAL/S and more [39].

Apart from the functional criticisms of HAL/S, there are major deficiencies relating to reliability. The language offers relatively poor typing (no programmer defined types, for example). The process communication mechanism, which relies on shared variables, is archaic and very error prone. It is not amenable to automatic checking for deadlock and similar difficulties. The control structures and expression structures of the language are also very poor. They are oriented more towards ease of programming than reliable programming.

2.3.4. Ada

Ada was chosen as meeting the DoD's specified requirements for a real-time language. It, like Modula, has gone through at least one redesign after public comment. These comments came in a wide variety. Some were objections to necessary features on purely aesthetic grounds, e.g. the ELSE within a SELECT statement was found "nasty", although it is needed for proceeding in the face of communications breakdowns or time-critical processing [40]. Some were specific suggestions about preliminary Ada which were included in "final" Ada, while others were disagreements other about whether it was easy to program a favorite

solution to some pet problem. We use the word "favorite" since it is often not the case that "you can't do X in language Y" but instead "you can't do X in language Y by method Z" [41].

Boute [42], in a study of preliminary Ada on representative communications control problems found it "very satisfactory", noting that the complexity and structure of the solutions matched that of the problem statement. On the other hand, Roberts, Evans, Morgan and Clarke [43], also looking at communications control and claiming experience in that area, say that the rendezvous mechanism is overly general and a potential time waster for message passing within or among processors. Specifically, a message that does not even need acknowledgement cannot be sent without at least four scheduling operations and that the sender is tied down until the receiver is finished reading the message. They state that Ada's philosophy is wrong for this application in that data rather than processes should be queued.

Mahjoub [44], also in the area of distributed processing, is more concerned with the asymmetry of the rendezvous. A task cannot know the sender of a message and messages cannot be broadcast. The concern with the asymmetric rendezvous seems to be a common one in resource allocation and scheduling [43,45], although there is a solution to this problem, involving creation of a resource task. An early problem [46] with scheduling was fixed in "final" Ada with task types so that manipulable structures of processes could be created. But problems with scheduling persist. Haridi, Bauner, and Svensson [47] and Mahjoub [44] favor static assignment of priorities by the user but, as we have noted, there are applications in which dynamic priorities are necessary. People

examining preliminary Ada [43] (before introduction of families of entries) found the rigid FIFO queue organization prevented urgent requests and tended to flatten different priorities to one level. Mahjoub [44] says that real-time programmers need to be able to write their own schedulers since different algorithms will be optimal for different applications. Roberts et al [43]. agree and declare that, to build a scheduler in Ada, one is building one scheduler on top of another, thus multiplying the overhead in what, in practice, is already a tight situation. Different applications have different ranges of speed requirements, some of the more highly constrained of which need radically different organizations. They conclude that Ada offers the wrong level of granularity of parallelism.

The method of inclusion of interrupt handling in Ada met with mixed response. Bennett, Kornman, and Wilson [48] and Haridi, Bauner, and Svensson [47] were in favor of it, but Mahjoub [44] was concerned with response time in that the handler task might not be scheduled right away or worse, might take a very long time to reach an accept for that entry.

The semantics of several Ada statements could result in bad states in a distributed system [44]. Between initiation and termination of an ABORT statement, a task might be able to communicate with another which, by virtue of being on another machine, has not been destroyed yet. Alternatively, a centralized knowledge base of what is alive and what isn't which had to be interrogated at every call would present a bottleneck which could easily bring a system down. The semantics have been revised in ANSI standard Ada to alleviate such situations [49]. Other potential overhead problems for real-time systems involve the

"Page missing from available version"

implementation level, the machine code insert capability was found useful [48] but dangerous if used unnecessarily.

There have been a few experiments and analyses of the potential efficiency of Ada implementations. Haridi, Bauner and Svensson [47] created a model intermediate language for Ada and ran (it may have been interpreted) programs hand-translated into it against a real-time version of C. The results of this experiment were deemed favorable for Ada's efficient implementation. Eventoff, Harvey, and Price [52] did an analysis of a generalized monitor based language vs. Ada's rendezvous on multiprocessor shared memory systems. They concluded that each approach was better suited for its own set of classes of applications. The monitor approach imposed less overhead for problems involving asynchronous communications and buffered synchronous communications while the rendezvous was better for problems requiring direct synchronization and problems which exhibited any degree of contention.

2.3.5. Summary

Programming languages have received a great deal of attention over the last thirty years and yet new ones continue to be designed. The reason is that no programming language yet devised is perfect. The design of languages is not a suitable problem for the short term, but the proper choice of an existing language to use is. There are many languages that are suitable for describing crucial software. Ada, HAL/S, and Modula-2 are examples.

The difficulties lie in finding a language:

a) which is of modern design,

b) which received sufficient care and analysis during its design,

c) which has a precise, formal definition,

d) for which compilers exist for the machines of interest,

e) for which validation of compilers and run-time support systems (within the current state of the art) is available,

f) and for which rigid configuration control of the language exists.

In the short term, these apparently minor issues are the really important issues. Differing opinions on what a language construct means, or subtle faults in compilers are major causes of faults in programs, but which have nothing to do with the programming language itself.

In practice, the only programming language which has faced all these issues and attempted to solve all of them is Ada. In addition, Ada is the only widely known and soon to be widely available language to include facilities for data abstraction. These facilities make the more modern design methodologies (such as Information Hiding, the Jackson method, and the Yourdon and Constantine system) far easier to use, and far easier for their use to be enforced. We conclude that Ada is the only choice of programming language for constructing crucial systems in the short term, and that language design is such a massive project that it is inappropriate for NASA to consider it. However, there are inadequacies in Ada and in the description of Ada. Short term investigations of the use of Ada and into its formal definition are appropriate in support of crucial software development.

Although we prefer Ada to the other extant candidates for programming languages for crucial real-time software, we still bemoan the fact that Ada was not designed with that purpose unwaveringly in mind. Ada, despite the original goals, was designed to do "everything for everybody". Hence, there are many aspects of the language which are not verifiable. Ada provides facilities which the community has deemed necessary to the creation of reliable software, but practices which lead to unreliable software cannot be prevented in any language with current technology without removal of features which are truly necessary.

2.4. Testing

Programmers have been running their programs against sample inputs to see if they "work" since the first fault was ever found in a program, yet no one has managed to move testing out of the world of ad hoc methods. The situation seems best described by the following quote:

> We know less about the theory of testing, which we do often, than about the theory of program proving, which we do seldom [53].

As long as humans are involved in the transformation of specifications of ideas into programs, we cannot be sure that no faults have been introduced without testing the resulting programs. The problem lies in choosing the set of tests which will uncover any faults in a given program. There are kinds of faults which we know about and can categorize, but there are also faults of a very much more subtle nature which are heavily involved with the semantics of the individual program and which we do not know any general way of detecting.

2.4.1. State of the Art

Despite some attempts [53,54], no one has yet completed a formal theory upon which to base the activities we call testing. Many of the proposed methodologies appear to be attempts to systemize the ad hoc methods of experienced program testers and to find systematic means of detecting types of faults which it is known that programmers commonly introduce. This may be in the hope that some formalism will fall out of such efforts and that an organized approach will help avoid wasted test-

ing effort in the mean time. Some directions concentrate on categories of faults while others tend to concentrate on the input spaces of the programs under test. One thing which must be remembered about testing real-time software is that one of the dimensions of the input space is time in that the behavior of the program usually changes over time for the same inputs. This complicates any testing strategy since the potential exists for, say, a program which reads two input values to require an infinite number of tests with different spacing of the inputs in time. Just when is enough enough? Statistically based reliability estimation and, of course, the exhaustive testing method, however, seem to be the major offers of a strategy for telling when to stop testing a given program [55,56,57]. Yet, there is a great deal of controversy within the reliability estimation camp about which basic theory of statistics applies, and exhaustive testing for real-time programs can be impractical.

There are known types of faults which seem to evade these efforts:

> With most testing methods, missing path errors are only detected by mere chance. In fact, missing path errors cannot be found systematically unless a requirements specification is available. A correct requirements specification would describe all the cases that should be handled by the program [58].

The allusion in the above to the unavailability of requirements specifications brings up a point of difficulty in testing. Due to the fact that in practice a program often reaches the testing stage without anyone having bothered to create a requirements specification, testers often have nothing but their own intuition to use in determining whether a program run against a test case has passed or not.

Another difficulty in testing is that programmers often try to "cover" themselves by including redundant conditionals in their programs. This fact often makes it difficult for a tester to determine whether a section of code is mistakenly unreachable or whether the conditions being evaluated are simply impossible. Further, it seems to be as difficult to create tests which create exceptional situations which the software is supposed to recognize as it is to create test cases which are intended to "stump" the software.

2.4.2. Contribution to Reliability

Without a formal theory, testing will only do two things for us:

a) It will assure us that, for the statistically meaningless set of inputs which we have tried, a program or system of programs "works."

b) It will give an unjustified increase to our subjective feelings of "confidence" in our software systems.

With a formal theory of testing, a set of tests performed in line with the theory would give a level of assurance of the program's correctness comparable to that given by a formal proof of correctness (without human mistakes in the proof). Short of a formal theory of testing, exhaustive testing (when possible) is by definition a proof of a program. Without a formal theory and with no possibility of exhaustive testing, the activities now pursued give a wholly unjustified confidence in programs.

2.4.3. Testing Techniques

In this section we give a list and brief explanations of the testing techniques which have been proposed in the literature.

a) Execute Every Line

Since it is impossible to have tested everything that a program does without trying each statement in it, it at first seems reasonable to create a set of test cases which together cause the execution of each statement in the program. This does generate a good number of test cases but it does not follow that executing each statement in a program exercises all of the program's functions.

b) Branch Testing

One of the ways functionality can be missed by simply executing every line in a program is for the program to contain a simple conditional branch around a statement, call it 'S'. The strategy of executing each statement would generate a test case which caused evaluation of the conditional to allow the statement 'S' to be executed, but would not generate the test case which took the other side of the branch. Branch testing is designed to make certain that each statement is executed and both possibilities are tried for each conditional branch in the program.

An example of a fault which Branch testing can miss is as follows. Suppose a statement being guarded by a conditional branch is supposed to be performed only under condition 'A' yet the program as

coded mistakenly allows the statement to be performed under either of conditions 'A' or 'B'. A test of both sides of the branch might be created containing two test cases, one in which 'A' was false, and one in which 'A' was true. If 'B' happened to be false in both test cases, we have a situation in which, although both sides of the branch would be exercised, the fault in the conditional expression would not be detected.

c) Path Testing

The idea here is to execute each possible path in the code as a method of checking the program's functionality. Executing each path is different from taking both sides of each branch. For example, if the code contains a loop for which it is possible to execute the loop 0,1,2, or 3 times based on particular input values, that loop contains 4 paths and thus requires 4 test cases. Should that loop be nested within a similar loop, the number of test cases required to test all paths in the loops is multiplied. Path testing cannot consistently detect paths which the requirements specification (if it exists) calls for but are missing in the coded program. For programs of a practical size, the number of possible paths approaches the size of the input space, so, to keep from testing forever, limits need to be made on loop executions or a closed form for loops needs to be proven to make this method practical.

d) *Structural Testing* (Also called White Box Testing)

The internal structure of the program as coded is used as a basis for choosing test cases. In using such test cases, the entire functionality of the program as coded is supposed to be revealed and that is to be compared with the specified requirements. Several other methods fall under this category. At one level the structure of a program is given by the conditional branches and call structure. However, one can also see a program's structure in other components. Geller [54] attempts to formalize structural testing.

e) *Functional Testing* (Also called Black Box Testing)

Functional testing attempts to test against the requirements specification for functionality. If the requirements specification states that the program should function in a certain manner when confronted with a category of inputs, it is tested with instances from that category. Test cases are chosen as if nothing other than the required behavior were known about the coded program being tested. It has been noted that this method cannot catch all faults since the method does not know anything about the coded program's internal structure i.e. the program may check out perfectly well but may behave properly only for the inputs used in the test and branch off into code which does something else entirely for other inputs.

f) Exhaustive Testing

All possible inputs are tested. One might think this impractical if possible, and improbable if (as in most cases) there is a large input space, but on current computers even the representable number of 'real' numbers is finite. With VLSI technology, it may become reasonable to create a large array of chips to generate test cases and run tests to exhaustion of the input space for a program. For truly crucial software, the cost of creating and running the VLSI chip array for years if it takes that long may be justified if a formal theory of testing is not found which can definitively give a more limited set of test cases for each program. Note that an exhaustive test of a program is by definition a proof of the program. All of the other test methods are capable of missing serious faults while the only problem with exhaustive testing is the large number of cases which must be run.

g) Error Seeding (Also called Mark-Recapture Testing)

In the error seeding strategy, a predetermined number of known faults are deliberately introduced into a program and arbitrary test cases are applied to the program (preferably by someone who does not know how many and what faults were seeded). At any point during testing, the percentage of seeded faults found is supposed to approximate the proportion of the naturally occurring faults which have been found so far by those tests. There is no reason to believe that the number of seeded faults is anywhere near the number of natural faults in any given program, nor that they occur

with a <u>similar</u> distribution. Also, the seeded faults would be manufactured by humans and as such would reflect the kinds of faults humans expect themselves to introduce. This biases the distribution of the seeded faults toward the first few kinds of tests the testing staff would try anyway. All of the seeded faults would be found quickly whereas the truly subtle and difficult faults would remain hidden.

h) <u>Statistical Testing</u>

Test cases are chosen via statistical sampling of the input space. Because real-time programs usually deal with the physical world, statistical testing is not likely to generate a realistic set of tests. Changes in the real world are smooth and gradual whereas a random sample from the input space is likely to vary widely.

i) <u>Error-Based Testing</u>

Experience with programming computers tells us that there are certain kinds of faults which we, as humans, commonly introduce. Error-Based Testing is an approach to testing in which test cases are designed especially to detect these kinds of faults. Unfortunately, we do not have a complete list of faults which humans can introduce, so no such set of tests is likely to detect all faults in a given program. Subtle faults are difficult to classify and more difficult to ferret out with classification-oriented testing strategies. This represents the brute-force approach to learning from experience.

j) Mutation Testing

Mutation testing derives from error-based testing, but as a methodology, seems to contribute more indirectly through evaluating the effectiveness of the test set than directly through testing the program. The required program and the coded program are thought of as being instances within a "cloud" of similar programs each of which differs from the others only slightly. The idea is to repeatedly transform the coded program P into similar programs P' by changing small parts of P. The set of test cases is run through P' to see if the test set is complete in its ability to distinguish between P and P'. If not, the tester must find a test which will distinguish outputs from the two. Each mutation transform is said to correspond to a class of faults. Among advocates of mutation testing, there seems to be a consensus that no more than a one "change" difference between P and P' is necessary to test the test set's effectiveness i.e. each P' is created via one small alteration to P. This method seems to call for combinatorically many more "runs" of tests than the size of the program being tested. It is difficult to tell how this process is supposed to determine whether the coded program P is the required program. For example, mutation testing cannot detect errors of omission where some part of the requirements specification is not satisfied.

k) Partition Testing

Goodenough and Gerhart [53] explain this and offer basic definitions and theorems which seem to be acknowledged as a good basis

for a formal theory. Briefly, the requirements specification is analyzed to determine a set of equivalence classes (partition) on tuples of the input space. Running as a test case one tuple from any equivalence class of the partition is completely equivalent to running as test cases all tuples in that equivalence class. Thus, exhaustive testing can be achieved by running one test case based upon one tuple from each equivalence class of the input space.

This seems like an ideal test method since it limits the total number of tests needed and is equivalent to exhaustive testing. The problem with this method lies in determining the equivalence classes. For realistic programs, this is not a solved problem.

1) Domain Testing

This is a refinement of Path Testing in conjunction with partition testing. Tests are devised to make sure that the set (domain) of inputs driving each path is correct, i.e. that the partition of the input space defined by the requirements specification and the partition of the input space effected by the coded program are one and the same. Some of the limiting factors are that this method with current technology cannot handle other than simple conditionals and that it cannot detect mutually canceling faults. There seems to be some merit in this approach as a lead-in to a testing formalism [58].

m) <u>Boundary Value Testing</u>

Test cases are created to exercise each conditional in the program as close as possible to the point where it changes between True and False. This is a limited form of component testing where the components are conditional expressions controlling branches.

n) <u>Range Testing</u> (Also called Stress Testing)

Same as Boundary Value Testing except the extrema of the ranges of values of each variable and input are exercised as well.

o) <u>Component (Unit) and Integration Testing</u>

Each component of the system is tested as a separate unit using whatever method is preferred, and test the combination of components (the entire system) for functionality as an assemblage of known-to-be-correct parts. Some people seem to have the idea that this can be done recursively.

Psychologically, we need to test our software before entrusting the safety of ourselves or our equipment to it. Practically, we find that the methods we use in testing are inadequate to the task. The hopes for formal theories upon which to base testing strategies worthy of our trust have not yet come to fruition, and may well never do so.

2.5. Programming Environments

A Programming Environment is the group of tools employed by humans to develop and, later, revise software. Much of the paperwork involved in such things as version control, data dictionaries, and management reports on programming projects can be considered drudgery and as distractions from the task at hand (building software). It seems reasonable to try to migrate that work onto computers as we have migrated much of the bookkeeping of programs to compilers of high-level languages. Although reasonable, this has seldom been done.

What work in the area has been done in the past seems to be disorganized and skewed toward the initial coding section of the software life-cycle. The reasons for this seem to be summarized by the following observations:

> The financial structure of many software producers is that production costs are a liability but maintenance costs are an asset or income ... In academic environments, using a portion of another person's code is often considered cheating. No credit is given for producing reusable software [59].

A few attempts, notably the Programmer's Workbench and the National Software Works, have been initiated to collect and implement on computers some of the tools which designers and programmers typically use [60,61,62]. More recently, with skyrocketing software costs (both in development and later revisions) and increasing complexity of systems, the DoD has become concerned about both automating the tools and integrating them. The DoD has commissioned the construction of integrated Ada Programming Support Environments (APSE's) [63,64] and the NBS [65], in the Spring of 1981, studied what can be done with today's

technology for medium and large software projects and what directions funding of research should take in the next five to fifteen years in the area of integrated environments for the entire software life-cycle.

We have had limited language specific computerized environments for years. Interactive BASIC, APL, and LISP systems have usually had their own file systems and editors, have been able to detect and notify the programmer about syntax and context-sensitive syntax errors as programs are entered. They have run-time systems capable of indicating errors in terms of source lines or statements. Some are capable of backing up, allowing source changes at execution time, and otherwise suspending execution while the programmer does other things, and have uniform and omnipresent sets of commands so that a programmer, for instance, does not have to "leave" the editor in order to "get" another file. A particular system, INTERLISP, has included many of the other capabilities and features to be described below [66,67]. Two of the primary qualities these systems have in common, and which are seen to be the enabling qualities of other planned environments, are that programmers deal with the systems interactively and that the tools in the systems "know" about each other and about the programming language.

Working from this starting point, much of the effort which has been put into programming environment research has gone into "smart" editors and source level debuggers [68].

Noting that we have been using text editors or other context-ignoring systems (e.g. CDC's UPDATE) to enter and alter programs in computers, and noting the success interpretive interactive systems have had in detecting errors as they are entered and the fact that the trend in

language translators has been toward syntax-directed translation, it becomes reasonable to consider syntax-directed, language-specific editors for entering program text. The use of such editors could eliminate compilations which are used only to detect and remove syntax errors. Since we are dealing with high-level languages, it is silly not to debug in terms of high-level language statements. Thus the idea of syntax-directed interactive editing is extended to source-level debugging in which one is able to interpretively "run" partial programs using such techniques as "stepping" through statements, substituting values, backing up, forcing branches, and making source-code patches while debugging.

As currently implemented [69,70], many such editor-debugger systems do not actually deal with a source code "file" but immediately internalize the input characters so that they use a data structure directly analogous to the syntax. In combination with CRT's they automatically "prettyprint" the source display as it is entered and might flag erroneous text in "reverse video" characters. Some even use color [71]. What does this buy in terms of reliability of life-critical software? We save syntax error debugging runs, and individual programmers on large projects can try out decisions in early stages without waiting for later testing stages when such decisions and any possible alternatives may have been forgotten.

The Cornell Program Synthesizer [70] allows a programmer to "hide" sections of code to abbreviate the source so all of the currently interesting parts can be displayed on a CRT at once. It also moves the CRT's cursor around from statement to statement on the screen as its

debugger "executes" them and presents a running display of variable-value pairs on part of the screen; the speed and direction of such "execution" can be controlled by the programmer and suspended or altered at any time. Statements are entered by selecting and filling in templates and errors are tolerated but flagged until corrected.

Work on improving the hardware of programming environments is being performed [72]. Noting that humans usually refer to several documents and several areas of a source program at once, this research is concentrating on how to partition screens and provide for multiple screens and still provide portable software and coherent, easily learned controlling commands as part of the command language of the environment.

Of course, the computerized tools already in use would not necessarily be abandoned. Since the editor would parse and internalize programs, a complete compiler is not needed. Rather we would need code improvers, code generators, and simulators for host and non-host target machines. The internalized form from the editor could also be fed into a static analyzer.

One ingredient considered essential to an integrated environment is a uniform command language. The language should be well human engineered with extensive help facilities which could, in advanced systems, even be anticipatory. The UNIX approach of keeping manuals online is seen as a large step in the right direction but has the failing that one must know (the name of) what one is looking for in order to find it. Although experimental systems are geared (consciously or otherwise) to be used by experts (their creators), actual environments must be able to serve novices with equal ease. The command language should

also be omnipresent: within reason, any command should be valid at any time. If the programmer is in the middle of using a tool when he remembers he needs to start up another, he should be able to call upon that other tool without abandoning the current tool, and the command language interpreter should be able to figure out the object and change of viewpoint without being explicitly told.

There has been considerable discussion on the degree of granularity or tool size and the amount of integration desired in an environment. Experiments with programming environments have ranged from the monolithic (a single gigantic program) such as INTERLISP [66], to the tool box approach provided by UNIX [73,74]. The monolith is seen as being less flexible and as hindering creation and inclusion of new tools provided by programmer-users. The tool box can be a jumble of bits and pieces so that a programmer must expend great effort just in picking out and properly composing the tools needed to perform even a simple operation. The trend seems to be toward small tools which can be composed, but for the environment to figure out which ones are applicable and how to compose them (i.e. for the tools to compose themselves), and for the environment to be easily told about and include new tools. There is also a strong trend toward having many tools running continuously as independent processes unseen by the programmer.

File systems and systems for keeping up with what is in each file play a major role in large projects. Does a given file contain a requirements specification, design specifications, source text, compiled binaries, executable code, implementor documentation, or test data for a given module? Where are all of the source modules for a given project

as they existed six weeks ago? Where is all material relevant to a certain paragraph of the system design document? Once a project gets to the stage of needing versions of modules, especially for revisions long after the original teams are gone, the picture gets even more complicated.

Most, if not all, environments involve a well-coordinated database. The database should manage objects (files), remembering properties about each one and managing relationships between objects with like properties and between objects whose properties exhibit dependencies. For security and information hiding in large complex systems, it should also provide and use access controls on its objects. All tools can be seen as creating new objects with properties relating to pre-existing objects. It is suggested that, in concert with the command language help facilities, the database could also serve as a kind of "Ann Landers" to field programmers' questions about policies, relationships among objects and groups of objects, and even "how to" questions to prevent people from constantly having to "re-invent the wheel", all the while avoiding violations of information hiding and security rules.

In heavily integrated systems, tools might monitor other tools' transactions with the database and initiate still other tools automatically when changes occur in objects which are related to other objects by dependency relationships. For example, a change and recompilation of an Ada package could trigger automatic recompilation of units which use it. Such tools might also insist that the original change be related to some report or test failure or try to aid the system documentation by obtaining some other sort of verification stamp.

Other tools (such as the Programmer's Assistant [75]) might be able to un-do a programmer's mistakes. This has implications for a database since a mere trace of a programmer's transactions is insufficient: the database must remember everything, all versions of all objects that ever existed for a project and their properties and relationships. A line of research arises here into how to compact this tremendous amount of information. One proposal is that, rather than keeping redundant information, the environment should keep a history of all objects and re-generate individual objects when needed.

Integrated programming environments are envisioned to have everything from the original requirements document in machine-readable form. Some distant prospects exist for specialized editors which "know" about the various kinds of documents in the system as the above mentioned smart editors "know" about programming language source. There is also the suggestion that such a requirements editor or design editor might feed into a quick prototyping tool which eventually might evolve into a program generator needing only a small amount of human "help" in the form of answering questions about ambiguities in the requirements specification.

Configuration management tools might monitor various releases of a system: who got it, did each recipient get all "fixes", etc. Such tools would track complaints, making sure someone handled them, and following them through changes and re-testing of modules and being sure the new configurations were actually released to the correct sites. All tests should be kept automatically by the system from the first test of a partial code segment on the source level debugger through to system

verification tests and the system should automatically re-run all of them (which are still applicable) as part of any change before release. In line with testing, there might also be automatic theorem provers whose results could be used to keep down the numbers of necessary tests.

One proposal would have management select a methodology or set of methodologies, based on the project's application domain, which thenceforth would drive the environment with respect to the project [76]. This suggestion is consistent with the goal of not having a particular methodology inherent in an environment yet guarantees that all programmers abide by management's rules. An environment could provide further management tools by automatically keeping track of who is working on what project/module, the amount of time and money being spent, and when the person moves on to something else. For instance, he might mark a module "complete" or signal that he has dealt with changes necessitated by some complaint or design, etc. change. The environment could also generate reports about these activities for purposes such as scheduling personnel and monitoring the progress of the project. Other reporting tools might include redundancy reporters and schedulers of review sessions based on some combination of elapsed time, percentage of the system that has been changed, and faults reported, etc.

An important consideration for environments for large projects is that often they are scattered over great distances and among many organizations. It has been proposed [77] that environments be designed flexibly enough to themselves be distributed with parts communicating with each other, or to adapt to dealing with other, perhaps manual, environments in a secure manner. UNIX has mail and news systems which can

serve for communication among individuals and groups at different sites or on the same site, but is considered by this proposal to be too general. A mail system is desired in which not just any text can be sent and in which receipt must be acknowledged and, for action requests, in which the acknowledgement must include an agreement or suggested alternative routing.

This has been an extremely brief survey of some of the things researchers are trying to do and are thinking about doing with programming environments. For more depth, it is suggested that one read [65], and for an analysis of the prospects of introducing efficient environments into the everyday world of programmers in the field, that one read [59].

The technology of programming environments as currently implementable [65] does not go far beyond collections of "good" toolsets appropriate to general software construction. The prospects for the future are brighter for well-organized, cooperating systems which may have a chance at enforcing adherence to those methodologies deemed more likely to produce reliable systems. Unfortunately, that day is not here. The current toolset approach has the same failing noted in the other areas examined in this section: The approach <u>allows</u> rather than <u>enforces</u> practices which may lead to the development of reliable software.

SECTION 3

Enhancements To The Conventional Software Development Cycle

3.1. Overview

The conventional process of developing software might be made more reliable through the inclusion of several advanced techniques and a controlled reorganization. Appropriating the prototyping concept from other fields permits rapid feedback from customers on the accuracy of the specified requirements and opens the producers' eyes to the problems which will present themselves during full development. Re-use of the work of others in the form of components limits the effort required in implementing and verifying a new system. An integrated environment can organize and enforce the flow of activities in the process, carry out some transformations itself, and provide the "memory" necessary for life-cycle-long configuration and enhancement control. Closing the gap between requirements specification and implementation languages through development of very-high-level languages (VHLL) would enhance the ability to simulate proposed designs before committing to them and would lessen the chances of introducing faults into design and implementation due to improper semantic mappings.

The cycle itself needs reorganization to place the decision-making and checking in the proper order and relegate to their proper roles less beneficial activities. Often, in the conventional process, implementation decisions are made during the design phase, no checking for design

validity is done until test cases are run on the implementing code, and the current state of testing methodologies is such that developers place unjustified importance on that part of the process. One proposal for automatically enforcing the ordering on activities in the cycle is embodied in the SAGA system. Here, a program enforces previously-defined rules governing which commands (such as "EDIT design document" or "COMPILE modulex") are valid at any given point in the development process.

3.2. Software Prototypes

In most engineering fields, a full-scale product is never attempted before a pilot plant or prototype version has been built and operated to the satisfaction of both producers and customers. This has rarely been the practice in software development projects. Software prototyping has grown out of experiences in which software systems have been completed only to be immediately scrapped because the customer realized too late that the product specified and built was not what was wanted [15].

Software prototyping technology is becoming a useful tool that should be pursued with a view to applying it to crucial software. The New York University implementation of Ada using the SETL system is a superb example of prototyping. The prototype implementation proved that Ada could be translated, to counter the arguments of those who could not design compilers for it. It provided early feedback to the language designers about things which were indeed unimplementable. And it allowed the Ada Compiler Validation Capability (ACVC) development to proceed in parallel with other translator development projects, since proposed validation suites could be tried out on the prototype translator before anything else existed. There has been substantial criticism of the NYU Ada translator because it executes very slowly. The critics are missing the key point that in this prototype, speed has been routinely sacrificed for functionality.

SETL is not particularly application specific although it is clearly more appropriate for prototyping compilers than control systems. Systems oriented to control systems' prototypes could probably be constructed on the SETL model.

An overview of the issues and work being done in software prototyping can be found in [78]. There has also been an NBS workshop on rapid prototyping a report on which is to be included in an issue of Software Engineering Notes in the Spring of 1983 [79].

To be reliable, a software system must conform to its requirements specification, but it cannot be built to meet requirements which are not known. A prototype model enables the customer to notice the absence of, and make explicit, requirements which had been assumed but not previously specified or the presence of things specified unintentionally. Large systems' requirements tend to change while they are being built. The early use of a prototype can serve to stabilize system goals sooner in the cases where changes to requirements were due to capabilities previously "left out" of the requirements specification. Often the originators of the requirements specification will not have experience with making explicit such things as a system's desired behavior. So it is difficult for analysis of the requirements specification to produce an accurate depiction of the behaviors wanted. Such problems can be ameliorated by allowing the eventual users to exercise a rapidly built prototype of the system. As in the above example of the SETL Ada implementation, a prototype may be used to experiment with possibilities for dealing with novel problems. Thus, production of a prototype serves as a means of verifying the transformation of the original idea to machine readable requirements specification. In that they will most likely build the prototype quickly while examining the original requirements specification, it gives the highest people on the production staff a chance to foresee some of the problems to be encountered later on.

Often, the most risky or uncertain aspects of a problem are placed into the prototype while more pedestrian aspects are ignored for the sake of cost savings on this version which will be thrown away. The analysis of what to leave out in the simple prototype helps to establish a basis for later application of functional decomposition. The Irvine report [78] offers several examples or real prototypes and the kinds of functions emphasized or left out to enable their rapid construction. One example involved the user-friendly interface and estimates of computational load for an automated FAA flight service station.

3.3. Software Components

A software component is a routine or set of routines with their own private data which have been written to provide a service useful to a variety of larger projects. A component which is in a portable form, and has been proven to actually provide the service it claims to provide, can be of great use in building crucial software. The persons responsible for the project can limit design efforts at higher levels to matching the components' interfaces. There is also less project-specific code to view with suspicion should faults be detected.

A limited form of software components has been with us for many years in the form of mathematical subroutine libraries. However, frequently other kinds of components have not been included in such libraries because of the difficulty of specifying what functions are performed and of writing understandable interface specifications. A more important reason is that, previously widespread languages which could interface to routines in libraries had to be able to access everything within a library; there could be no information or auxiliary routines hidden from the user. Further, the desire for highly optimized code has led to users' reluctance to use anything they did not tailor to individual applications.

These otherwise valid reasons do not apply to Ada. The Ada package mechanism can provide portable abstractions of higher-level concepts and structures in which external interfaces are fully specified yet with internal workings inaccessible to users. As for the optimization problems, to make the language usable in real-time Ada compilers must perform extensive optimizations including those which apply across

procedure boundaries anyway. "Optimization" via algorithm selection can be done by providing a set of components for each function with additional specifications describing the types of situation most likely to benefit from each component in the set. These qualities are not specific to Ada. The technologies were not available in earlier languages or had not all been brought together in one system before.

One system for using components in the development of efficient and correct software is described in [80]. The view taken in that system is that a software component can be seen as a part which itself is composed of parts depending on level of abstraction. The traditional or craftsman approach, through an expensive and time-consuming process, produces efficient software requiring custom "maintenance" in the same way as any "hand-made" item does. The parts-and-assemblies or components approach produces cheaper software with a common "language of discussion" and allows the parts to be studied for the ways in which they can fail and be repaired in all applications. The component approach does not eliminate the craftsman since he is needed to build good, reusable parts, and a system can rarely be built entirely from such reusable parts. The relative costs of the approaches depend on the numbers of like programs to be eventually produced. Since components represent implementation choices, a fully coded and compiled part cannot be seen as an assembly which can be optimized in a manner which would make software components usable directly. Thus that system represents components as designs or input/output specifications and enabling conditions which can influence the choices of an automatic coding system in optimizing for particular applications. In that system, libraries of components were built for

specific application domains but were able to make use of components in libraries for other domains which had already been built. The components contained several alternatives with individual enabling conditions so that an alternative could be chosen based upon "goals" for development as specified by a human interacting with an experimental transformation system. The components were relatively small but the system could build up larger programs by combining them and using some components for selective replacement within the text of other components. The author describes this as "A domain's software components map statements from the domain into other domains which are used to model the objects and operations of the domain ... Each object and operation in the resulting program may be explained by the system in terms of the program specification." The examples actually presented in the text are necessarily small and textually oriented, but include the construction of a natural language parser-generator and a natural language relational database.

3.4. Integrated Environments

Environments were discussed in Section 2. However, as was noted, the current state is not as advanced as it could or should be. Much research needs to be done to create programming environments which actively take part in the production and correct revision of software systems rather than passively offering individual unrelated and inadequate tools. This active participation characterizes the concept of an *integrated* environment. Where the environment knows about the forms and processes of software projects in general and of an individual project in particular, it can institute and impose appropriate checks and documentation policies. Thus, the minimum amount of human work/ineptness need be applied in a system's development.

The idea of environments needs to change from the box of tools approach to active participant. An integrated environment needs to recognize and save potential components for future use, and recognize places where a previously developed component can be used and insert it. The environment also needs to be able to generate a prototype model from any level of "document" such as requirements specification, design language "program", or partially implemented software to allow exercise by users or simulation at any stage of the project. The order of events needs to be controlled and enforced. For example, discovery of a fault should trigger re-examination of the requirements specification before design, and that before implementing code.

The HOST [81,82] system is to be such an integrated environment. By using H-Graphs as a standard form for internal manipulation, all of the system's "tools" can deal with the *semantic* basis of the project.

Thus the project's requirements specification, design language, and implementation language can all be reduced to H-Graphs, or an H-Graph form can be entered directly. This allows comparisons for compatibility and consistency among all forms of the software, and the use of components developed for other projects, perhaps in a different language. Finally, prototyping can be achieved via interpretation of the H-Graph representation of the requirements specification.

3.5. An Improved Conventional Software Development Cycle

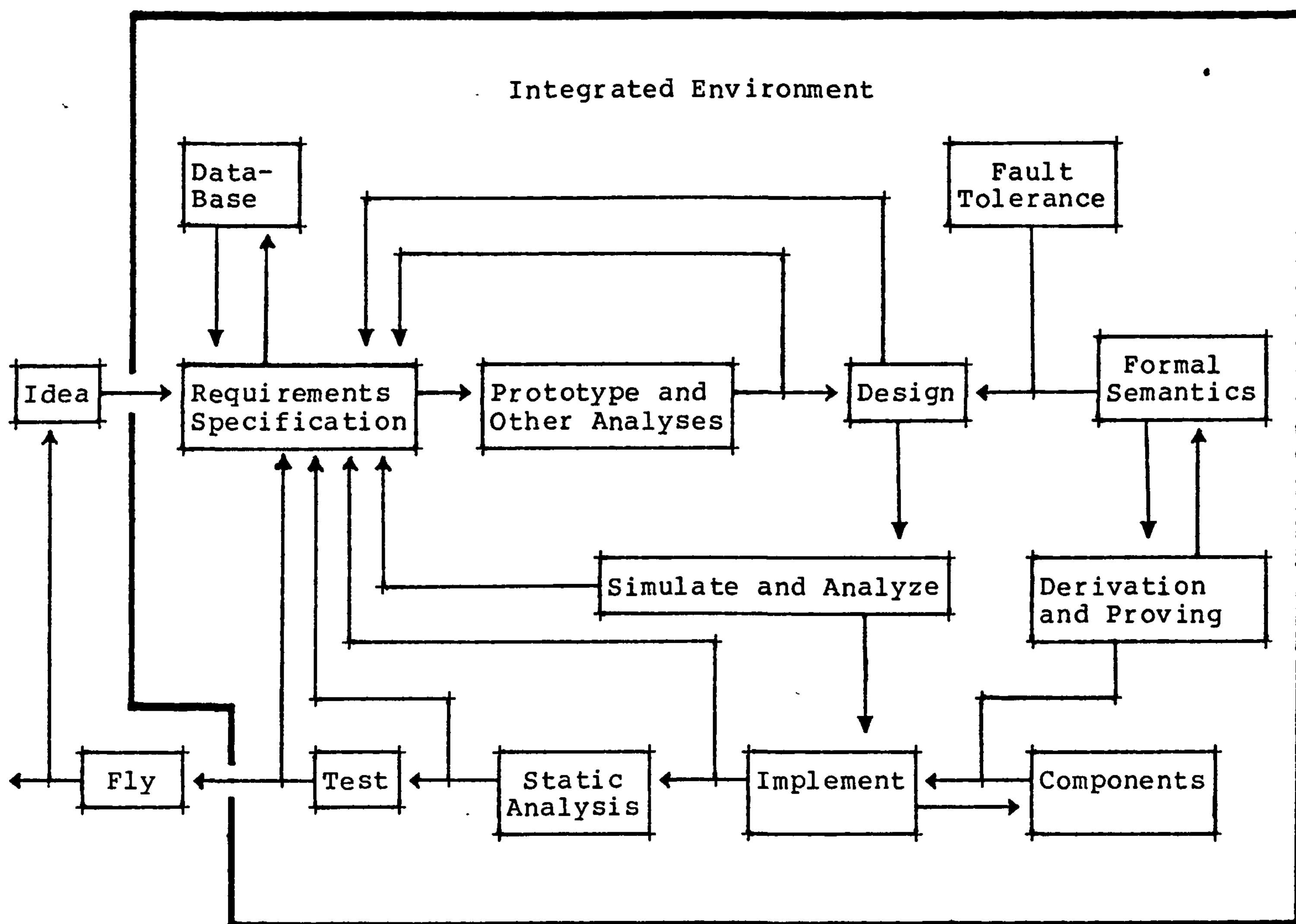
We propose an enhanced software development cycle to include all the techniques mentioned above. It is shown in Figure 3.1.

The entire process is controlled by an integrated environment which, among other things acts as a determiner of the "next valid activity". All of the tools interact with a database which is pervasive, supplying the appropriate information where needed. The database is made explicit in the figure at the interface between the idea and the requirements specification for two reasons.

The first is that during post-delivery, as needs change, additions to the original idea can re-enter the system in the process normally termed "maintenance" (more accurately called "revision"). The original idea and requirements specification are retrieved from the database and fed through a consistency checker along with the additions. The consistency checker should insist that conflicts be explicitly overridden. (As a part of configuration control in the environment, the original requirements specification is not overwritten but a new one for the revised project is created.)

The second reason that the database is made specific is that during original entry of wants, previous projects' ideas can be compared and suggestions made for clarifications. Also the consistency checker can play a part in amendments during a project's development. The database as described resembles what has often been called an "expert system", and it is intended to be at least a primitive version of one.

All but the final path in the figure lead back to the requirements specification. Any fault detected during testing, analysis, or exercise



Improved Conventional Software Development Cycle

Figure 3.1

of a prototype and any item found unimplementable during coding or design must be traced back to its origin in the requirements specification as prime suspect with other areas becoming suspect if the requirements specification is found innocent.

Prototyping follows the requirements specification since we must have some specification from which to build the prototype no matter how vague. The prototyping step occurs between the requirements specification and design on every iteration. Any revision of the requirements potentially invalidates the previous requirements specification and its approval which was derived from exercise of a prototype. This can be seen as an instance of needing a rapid prototyping capability, and if the prototype can be automatically generated or revised in real time during the human-expert interaction that would be even better. In the case of a fault being found and the requirements specification being found innocent, the prototyping and analysis box may only entail the check that the requirements specification really is correct.

Note here that the figure represents a general plan and the details of each box may be complex with internal path control directed by the environment and with the amount of complexity dependent upon the particular methodology chosen. For example, from Section 2 we saw several requirements language-analyzer pairs and several design methodologies, a choice of any of which would radically alter the appropriate box over the others.

A prototype can be viewed as a model or simulation but we distinguish between the vehicle for "verifying" the mapping of ideas to requirements specifications and that for verifying the mapping of

requirements specifications to designs, just as testing and static analysis are distinguished as steps verifying the mapping of designs to implementations. A device for running a test case may be termed a simulator, but more accurately a target environment emulation device.

Fault tolerance is brought in as early as the design phase. Such capabilities should be designed into a system rather than appended after full development. We have more to say on fault tolerance in Section 5.

After a design has been verified through simulation or other analyses, its implementation should be made formal wherever possible. This includes formal derivation where the state of that technology is applicable, the use of previously verified components, and proving of coded portions of the program to the maximum extent possible. A formal semantic definition of the implementation language and a formal semantic representation of the design can be used to direct and guide the implementation process to the extent that matching semantic definition languages may allow the design "document" to select statements or routines.

Note that the components box involves some give and take with the implementation box. Design information and specified requirements information influence the choice of components from the database and, in a system such as [80], influence automatic optimizations on the components chosen. Recognition, automatic or otherwise, of newly created items in a project which could themselves be used elsewhere as components can be made to trigger inclusion of these items into the components database.

The problems with testing were described in Section 2. However, any exercise of a program has the potential for detecting some fault, and humans have a psychological need for some sort of testing of any newly developed product. If and when a formal theory of testing is developed, the figure has reserved a place for it.

The stages of the software life-cycle are often said to be conceptual, actually taking place in parallel or in an overlapped manner. The controlling environment should insist on an order which separates the concerns of requirements specification, design, and coding. Just as separation of concerns in a design limits complexity and enables an accurate mapping of specified requirements, separation of the stages in the development cycle limits the amount of information needing to be dealt with at one time and prevents premature decisions in one part of a system from having undue influence on the rest of the system. The fact that we have paths back to the requirements specification does not change this position. Any alterations to requirements specification, design, or implementation necessitated by traversal of such a path should cause the replacement of the affected parts, no matter how widespread.

SECTION 4

The Inadequacy Of The Software Development Cycle

Given the stringent reliability requirements of crucial software, can the conventional software development cycle or the cycle described in Section 3.5 be used to build software of the desired quality? The answer of course is yes, but rarely and unpredictably. There may be circumstances in which reliable software is developed using conventional methods. The problem is knowing that the software is sufficiently reliable. Crucial applications will certainly not use software which contains known faults. However, what is required is assurance that either there are no residual faults or that the unknown number of residual faults will not lead to failure. We claim that the conventional software development cycle cannot meet these requirements. We will attempt to justify this claim with some experimental evidence and expert opinion.

Two crucial applications relying on digital computers are the control of manned spacecraft and the control of nuclear weapons. Software failure in either case could be catastrophic. Both applications presently rely on conventional software development methods, and both have experienced failures in production software systems. For example, the first launch of the Space Shuttle was delayed for two days [36] by a software fault. Fortunately the consequences were not serious. In another example, the launch control system for the Trident missiles on

board a Trident submarine went into an infinite loop when an operator attempted to "launch" all 24 missiles in sequence during an exercise [83]. All the missiles were disabled, and had this been a battle situation, none could have been launched. The diagnosis of this problem was operator error since the missiles are supposed to be launched in three sequences of eight each.

The software which operates the SIFT computer [84] must be regarded as crucial since the correct operation of the computer relies on the correct operation of this software. The designers of SIFT did not use the conventional software development cycle but chose instead to use a formal verification method. They feel that faults were found this way which would not have been found by conventional methods [85].

The software which supports communications of classified data is crucial in the sense that failure might allow compromise of classified data. Note however that failure which causes loss of service is acceptable provided security is maintained [86]. This is far less stringent a requirement than is imposed on crucial software.

The workshop on The Production of Reliable, Flight-Crucial Software [87] was asked to discuss the issues involved in crucial software development and make recommendations on research areas which should be pursued. The first conclusion reached (which was agreed upon unanimously) was:

> There is serious doubt that it is presently possible to produce flight software systems having the stated level of reliability and to assure that they have that level of reliability.

Finally, Winograd [88] has argued the case for major changes in software development methodologies for various reasons, and Wasserman et al [89] have pointed out that, in crucial applications, the consequences of software failure may extend beyond the normal concerns for human life or expensive equipment to legal actions against the programmers involved.

Taken together, these points convince us that the conventional software development cycle is inadequate. There may be examples of programs running which have been created by conventional means and which appear to be reliable. The key word here is "appear". It is necessary to show scientifically that the software is sufficiently reliable.

SECTION 5

Fault Tolerance

Although fault tolerance has been applied extensively in hardware, it has received relatively little use in software. There is an important distinction between hardware and software faults which must be must be born in mind in discussing fault tolerance. The majority of hardware faults are the result of physical degradation of components whereas software faults have the characteristics of design faults. This precludes the use of parallel executions of identical software to guard against faults but, in contrast to hardware, software does have the potential for being permanently fault free.

In this section we review the state of the art in software fault tolerance. We assume the reader is familiar with the basic principles of the various methods. In general we feel that software fault tolerance has the potential to increase reliability dramatically. It can be considered part of the conventional software development cycle. It is considered here in a separate section because of its importance.

5.1. Recovery Blocks

Recovery blocks were proposed as a technique for providing tolerance to faults in sequential programs. A very strong theoretical background has been developed for recovery blocks. Provided erroneous states are detected, damage assessment and state restoration are totally reliable, and continued service can proceed from a secure starting point. Two disadvantages are the need for hardware support (the recovery cache) for state restoration and the fact that this is backward error recovery. Despite the fact that the recovery cache was patented ten years ago, there are no commercially available machines with recovery caches and so there is no opportunity to use recovery blocks in practice. Backward error recovery could be a problem in real-time systems and has to be taken into account.

Attempts to extend recovery blocks to concurrent programs led to the problem of the domino effect and to the conversation technique as a solution. Conversations are theoretically quite simple but rather surprisingly no syntax has been chosen for their inclusion in programming languages (in contrast to recovery blocks). Some proposals have been made [90,91] but none has gained even modest acceptance and none has been implemented. The reason for this situation is twofold. Firstly, although conversations seem simple, integration of their semantics into a language supporting concurrency is a major effort. Concurrent languages are still in their infancy and there are many very difficult issues in their design. Incorporating conversations just makes a very difficult problem even harder. The second difficulty with conversations is that once again hardware support is required. In

contrast to recovery blocks however, in order to implement a conversation, a recovery cache is required for _every process_ involved. Thus in principle, many logically separate cache's have to be provided.

It has been observed [92] that many real-time systems have properties which allow fault tolerance using backward error recovery to be included fairly easily. A framework has been proposed which allows fault tolerance to be included in _cyclic_ real-time systems with no special hardware provisions. It has been pointed out that this work does not cater for real-time systems which are interrupt driven and this is a serious weakness. The work is being extended to include interrupt driven systems to provide a comprehensive approach to fault tolerance in real-time systems.

5.2. N-Version Programming

With obvious analogy to hardware techniques, N-version programming [93] has been proposed as a method of providing software fault tolerance. It relies for all aspects of fault tolerance on the execution of multiple versions of a program and comparison of their results. This is somewhat weaker theoretically that recovery blocks. Damage assessment is handled by the assumption that damage will be limited to the versions in the minority when the vote is taken. To ensure that this is true, the versions must be physically separated. Clearly this is not easily achieved for parts of programs such as subroutines. In practice, this limits the application of N-version programming to the system level and precludes its inclusion in technologies like software components.

A further difficulty is the treatment of state restoration. Again, this is handled by the assumption that the different versions do not interfere and that the states of the versions in the majority after the vote are consistent and ready for continued service.

It is important to note that any versions in the minority after voting must be assumed to have failed. Thus they cannot participate in any further system activities. If the system is required to continue operation, there must be sufficient versions remaining for voting to be possible.

Voting presents another problem for N-version systems. If the versions are implementing some form of arithmetic, the results may not be in bit-for-bit agreement. In such cases, have there been failures? Probably not, but to avoid detecting failures in these cases it is

necessary to use ranges rather than exact inequality tests. How wide should the ranges be? If they are too wide, failed versions will not be detected, and if they are too narrow, successful versions will be rejected.

An advantage of N-version programming is that it can be readily applied to concurrent and real-time programs since it does not rely on backward error recovery. Indeed, it has already been applied to a crucial application [94]. Hardware support is required for N-version programming in the from of provision for physical separation (usually multiple processors) and for voting.

5.3. Reliability Improvement

A major area of concern in all aspects of software fault tolerance is a lack of data showing that reliability is improved by using it. No major demonstrations have been performed which show that fault tolerant crucial systems can be built (although one such experiment is underway at the University of Newcastle upon Tyne in England [95]), let alone that they will be adequately reliable.

It is intuitively reasonable to expect software reliability to be improved by using software fault tolerance. Intuition is often wrong, and it is necessary to resolve the remaining issues in the technology of both forms of fault tolerance and to obtain reliable data on reliability improvements that can be expected before the technology can be recommended for inclusion in crucial software.

SECTION 6

Verification

By verification we mean the technology of establishing a mathematical proof that an executable computer program complies with its requirements specification. We have not spent a great deal of time on this topic because of the substantial experience already in Langley's Fault-Tolerant Systems Branch. The SIFT project and the contact with the SRI verification group is extensive and provides a far better assessment of that technology than we could obtain from the literature. For the sake of completeness, we have included an extensive bibliography on verification.

We make several observations of a cautionary nature because we feel that it is important that verification not be viewed as a panacea. First, if a program is to be proved, it's requirements specification has to be in machine readable form which is amenable to analysis and this is not always easy. For crucial applications it could be required but that means that the engineer and the computer scientist will have to communicate in an informal language (English) or the engineer will have to learn (and be comfortable with) the formal notation. Another difficulty with verification is the complexity of the proof process. Theorem provers are a help but there is still a need for human guidance and inspiration. This makes the proof process long and tedious, and contributes to the fact that program proofs are not a routine matter and

proofs of programs more than a few hundred lines long are very rare.

Perhaps the biggest danger with verification is the prospect of the proof being wrong, i.e. a proof being produced for a program containing faults. There are numerous examples of this in the literature. One example is by Geller [54] in which two proofs are presented for a program which is wrong. It must also be noted that there are major areas where verification has had no success whatsoever. These areas include floating point calculation, concurrent programs, and until recently real-time programs.

Despite these reservations, there have been some remarkable successes in verification technology. The proof of a simple real-time program [96] is very encouraging. The recent proof of a program that is more than 4000 lines long is also a major accomplishment. This program and its associated proof were constructed at a measured productivity rate of four lines of code per programmer per day [97]. This compares very favorably with the productivity obtained using conventional methods.

Provided the problems are kept in mind, verification appears to be a technology that is almost ready for application in some parts of crucial systems. The comprehensive approach to crucial software engineering that we propose in Section 8 incorporates verification.

SECTION 7

Automatic Programming

7.1. Introduction

We have come to the conclusion that in the long term, major improvements in the reliability of software will only be achieved if the ad hoc methods of construction in which humans are involved can be eliminated. The problem in the classical software development cycle, or any enhanced version of it, is that humans make decisions at every stage and thereby introduce errors at every stage. As noted in Section 6, verification can provide substantial reliability improvements. It relies, however, on human-generated programs and human-generated proofs for the most part, although there may be extensive computer checking. Despite impressive success with verification, it is only an intermediate step. The long term goal has to be the removal of unchecked (or uncheckable) human decision making from the software generation process.

The creation of a software requirements specification is the only step in software development where human decision making is required. It is the link between the "idea" or "concept" for a system which exists in a human's brain and computer processing of that idea. Once a complete, formal requirements specification exists in machine readable form, it is amenable to many formal methods of analysis. In principle, these methods can be used to build an executable computer program directly from the requirements specification with either no human

intervention or just human guidance. Thus it is potentially possible to derive a program from its requirements specification and thereby "prove" that the resulting program complies with its requirements specification (this is the definition of reliability). Note that no proof in the classic sense of program proving is needed. Where formal methods do not yet exist, or are not yet sufficiently powerful (such as program design), additional research can be expected to yield satisfactory new or improved techniques.

Unfortunately, since the requirements specification is the first machine readable version of the "idea" or "concept", the translation from the "idea" to the requirements specification cannot be automated and subjected to completely formal methods. Thus it will never be possible to prove that the requirements specification corresponds precisely to the original "idea". Many faults are introduced because of the necessarily informal (and thus inadequate) translation of the "idea" into a requirements specification.

The ideal situation would be one in which the requirements specification is entered into a computer by a human at the highest practical semantic level and the process of producing an executable program would be left to the computer. The only testing that would be needed would be that which convinced the human that _the requirements specification as initially entered_ corresponded to the "idea" in his/her head. Emphasis must be placed on notations which allow requirements specifications to be expressed in a form where the semantics can be determined by processors which will be responsible at least for analysis and possibly for constructing the executable program.

As noted above, no proof of correctness would be needed for programs which are automatically derived from their requirements specifications. Similarly, no fault-tolerance methods would be required or desired. Another major advantage of this approach is the simplified procedure needed for enhancement or modification. Changing a program should always begin with changes to its requirements specification though it rarely does. Changing a program may involve a substantial redesign of algorithms and data structures although since this is so time consuming, quicker methods involving "patches" are often used. For programs which are automatically derived from their requirements specifications, these problems go away. The requirements specifications have to be changed but the rest of the process is automated. Although it may require very large amounts of computer time, the derivation can proceed automatically.

The practical implementation of these notions is termed automatic programming. We have been reviewing this technology at some length in order to determine its feasibility in the long term as a method for building crucial software. A modest version of the technology has been in use for some time in the form of high-level languages. Programs written in high-level languages are really requirements specifications for machine-language programs. These machine-level programs are not written by humans but are derived automatically from the requirements specifications by a computer program; namely a compiler. It is not unreasonable to state that most programmers never write programs; they write the requirements specification in a non-executable language (a HLL) for a machine language program which is synthesized automatically.

This application of automatic programming is readily accepted and needs to be extended to higher-level constructs to allow more of the translation process to be automated.

The state of the art in automatic programming at the level needed to eliminate human programmers from all but the requirements specification phase is very far from practical use, as would be expected. However, we have performed an extensive literature search, and the example systems that have been built and reported are quite impressive. For example, with a little human guidance, a program to solve the eight queens problem has been derived from its requirements specification.

There are several approaches to automatic programming and there are related research projects which contribute to the goal of eliminating human creativity from programming. There are many excellent surveys of this field and they will not be duplicated here. The interested reader is referred to the bibliography section on Automatic Programming, to the survey by Biermann [98], and to the survey in the Artificial Intelligence Handbook [99]. These latter two papers are excellent surveys of the state of the art in automatic programming. Both are quite long (63 and 110 pages respectively) and summarize the theory behind the methods as well as describing the major operational systems. They are both very readable and the second one is very recent (1982).

In this section we will discuss briefly the various approaches to automatic programming and the associated technologies. Although under the general heading of automatic programming, two related research projects are mentioned. They are SAFE and the Programmer's Apprentice. These two systems are applications of artificial intelligence which help

reduce human error but are not complete program synthesis systems.

7.2. The Issues in Automatic Programming

Any automatic programming system will require as its input a program's requirements specification. We have already noted substantial difficulties in this area in the conventional software development cycle. Precision, freedom from ambiguity, and so on are all very hard to achieve. For automatic programming, the situation is more difficult since the requirements specification has to be amenable to machine analysis. This seems to eliminate natural languages which are very convenient for humans but very difficult for computers to process. The predicate calculus is usually suggested as a suitable notation, but it is quite difficult for most humans to deal with.

This conflict has lead to two important lines of research. One is automatic programming systems based on the predicate calculus [100], and the other is an effort to build a processing system for the English language [101]. In limited ways, both have been successful and we recommend the products of this research in our comprehensive approach (see Section 8).

Another technique for specifying requirements is the use of examples. The intent is that the user gives the system examples of the computation required and the system builds a program which satisfies all the examples. Two different approaches to defining examples are used. In one approach, the output expected for each input is given. In the second, the user works through the desired algorithm with sample inputs and the system is required to infer the algorithm.

Programming by example has been studied in depth and implemented in two commercial systems by IBM [102,103]. Unfortunately, a program which

works for all the examples may fail on the first application. Specification of requirements by example and hence programming by example does not seem like a viable technology for building crucial systems and will not be discussed further in this report.

The output notation used by an automatic programming system is called the target language. Different systems use different languages but in most cases the target is some kind of high-level language. The output of an automatic programming system can be translated into an executable program by some form of compiler, thus completing the synthesizing process.

In principle, it is not necessary to be aware of the existence of the target language or the fact that one form of the desired program is written in this target language. In practice it is important and quite useful. We have noted that automatic programming may not be able to build programs of the size or complexity that we need. In section 8 we propose an approach in which part of the program is synthesized automatically and part is written by conventional means. The parts will have to be merged and this can best take place at the level of the target language.

To reduce the complexity of program synthesis, most existing systems restrict the application area that they deal with. Since automatic programming is basically a research area, this approach is appropriate. The goal of most researchers is to develop algorithms which will synthesize something rather than something specific. Thus existing systems are impressive (in some cases) but not particularly relevant to crucial software development. This means that it will probably not be possible

to use any existing system, or minor variant thereof, in a practical crucial software development system. It also means that it will probably not be possible to use a single automatic programming system, even if it were specially developed, in crucial software development. In practice, it will probably be necessary to use several complementary systems; each working on part of the problem.

There are several fundamentally different methods of operation used in the various automatic programming systems. The different methods have various advantages and disadvantages but externally the major difference is in the size of programs which can be synthesized. At the time of writing, the program sizes vary from a few lines in the approach of Manna and Waldinger [104] to many tens of lines in the system of Balzer105 . It must be kept in mind that there are other issues to consider in comparing systems. For example, the Manna and Waldinger approach is potentially more general although though there is no clear limit to the transformation approach of Balzer.

7.3. Automatic Programming Systems

In this section we discuss some of the more significant automatic programming systems and consider their relevance to crucial systems. This is not a general survey. The interested reader is referred to [99].

SAFE

The SAFE (Specification Acquisition From Experts) system [101] is an attempt to build a program which can interact with a user in natural language in order to determine the requirements for a program. Its input is in a subset of English and its output is a requirements specification written in a formal notation. Thus SAFE is trying to solve the difficulties known to exist in the phase of software development where requirements are specified.

SAFE expects its input to be ambiguous and incomplete. The goal of the system is determine these problems and resolve them by interaction with the user. With this goal, and the apparent successes of the system, it appears to be an ideal candidate for use in crucial system development.

The literature on SAFE indicates that its primary focus is on requirements specification. In fact in recent work [106] the SAFE system has been coupled to a system supporting transformational implementation and a complete automatic programming system is being built. This was the original goal of the SAFE system designers. SAFE is just an intermediate step but a very important one.

Programmer's Apprentice

The Programmers Apprentice is described by its designers [107,108,109,110] as an intermediate point along the road to the desired goal of automatic programming. It is not an automatic programming system. It is an application of artificial intelligence methods in a system designed to help a human programmer by checking his work. It has a substantial "knowledge" of programming maintained in a library which it uses to help validate that human generated code is consistent with the specification for that code.

The system is intended to operate interactively, conversing with a programmer as an apprentice would. The examples of the system in use which appear in the literature are very impressive but apparently describe _what it would do if fully implemented_ rather than what it is able to do as currently implemented. It also appears that the system has not been the subject of active work for some time.

The ideas behind the Programmer's Apprentice and the capabilities that it apparently provides seem very suitable for use in crucial system development. This technology is probably applicable in the short term.

PSI

PSI is a system built at Stanford by Cordell Green and colleagues [111]. It is _very_ large and apparently powerful. Some confusion can easily occur in examining the literature since two major parts of PSI (PECOS [112] and LIBRA [113]) have been described in separate papers and appear to be separate systems. PECOS and LIBRA are both capable of some independent operation but are basically parts of PSI. In fact, PECOS is the "coding expert" and LIBRA is the "efficiency

expert" of PSI. We will not discuss PECOS and LIBRA separately.

As far as we can tell, PSI is the most comprehensive automatic programming system that has been built and it is the only system to have addressed certain issues. For example, its method of operation is basically transformational but it can reach a state in which several different transformations are valid. Note that any system based on transformations can reach such a state. A choice at such points could be made by a human, or by the system at random, but PSI invokes its efficiency expert (LIBRA) which searches the valid transformations for the one which will yield the most efficient program. Efficiency could be terms of time or space.

PECOS is the "coding expert" for PSI and uses a database of rules to make decisions about program synthesis. An impressive aspect of PECOS is the way in which certain simple, "well-known" rules of programming are contained it is database. For example, the following rule which is frequently used by programmers can be included in the database:

> If a collection is input, its representation may be converted into any other representation before further processing.

In papers describing PSI and its subsystems, various examples are given of programs that have been synthesized. Although we are impressed that anything can be synthesized, we find the semantic level of the input to be very low. The input definition of the problem in many cases seems to contain too much detail and in fact is virtually a complete program.

7.4. Conclusions About Automatic Programming

Several experimental systems have been built (many have not been mentioned here but see the bibliography), and a great deal of research has been performed on automatic programming. Though the subject is far from ready for general use, it does hold great promise and there do not seem to be any fundamental, theoretical reasons for thinking that progress will not be made in automatic programming.

The advantages of synthesizing programs from their requirements specifications are many but the most important here is the potential increase in reliability. This approach seems to be the only viable way of ensuring that programming is carried out in a scientific way, and does not rely on human fallibility and the associated introduction of faults.

SECTION 8

A Comprehensive Approach

8.1. Overview

Recall from Section 1 that no effort is made to quantify reliability or reliability improvements in this grant. Thus in this section we describe an approach which we feel will produce the most reliable software product but we make no claims as to the degree of reliability which might be achieved.

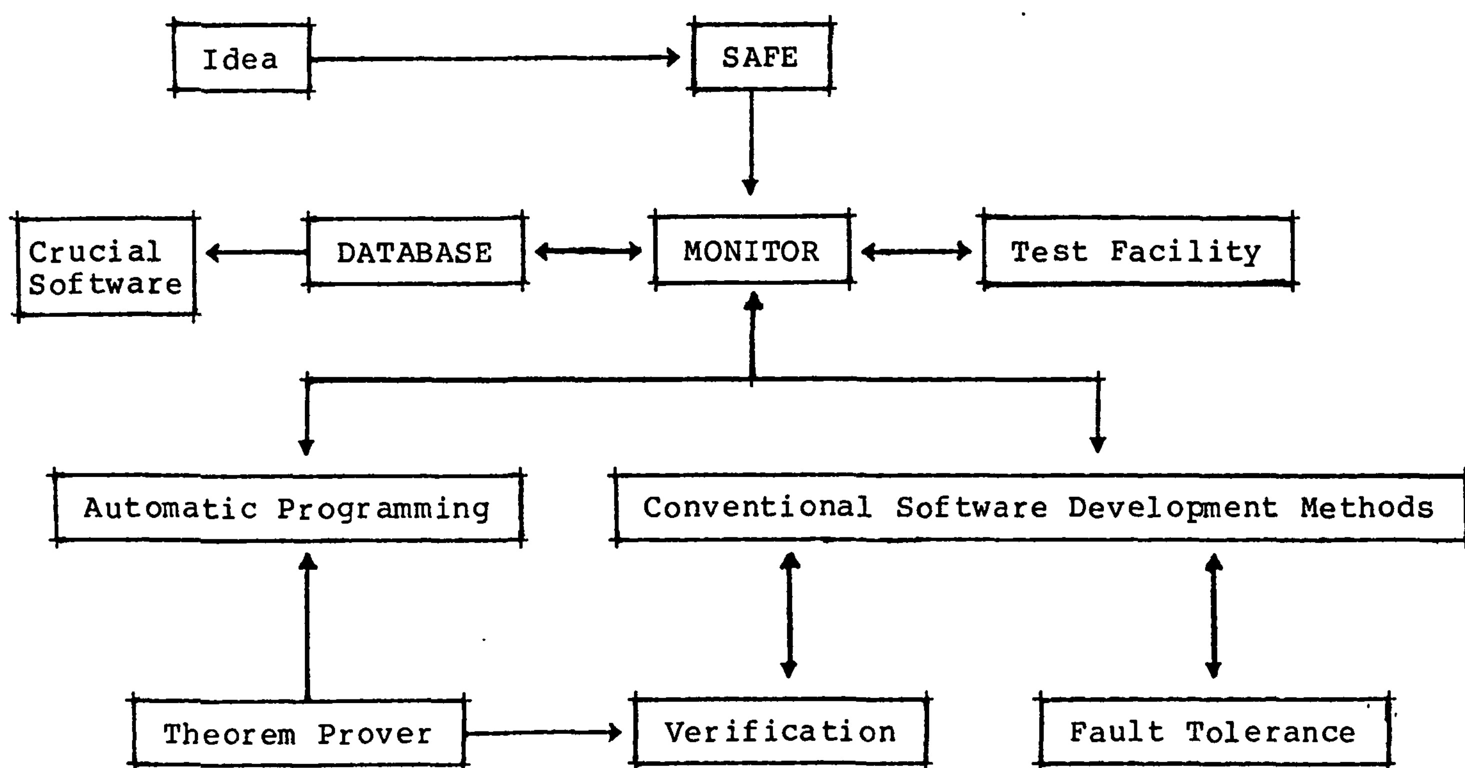
Given the inadequacy of the conventional software development cycle, the difficulties with verification, and the infancy of automatic programming, how should crucial software development proceed? Moving from conventional methods to formal verification will yield an improvement in reliability of software. Similarly, moving from formal verification to automatic programming will yield another improvement, and automatic programming probably represents the best that can ever be done. An ideal solution would be the use of automatic programming for the entire development of software for crucial applications. This ideal is far from possible at this point so a less desirable, more practical approach must be sought.

Basically, we propose that a combination of these three techniques be used. For a given application, those parts which can be synthesized by an automatic programming system should be. Of the remainder, those parts which are written by humans but are amenable to verification

should be verified. The remainder of the application (which may well be large parts of it) will have to be built with conventional methods. To provide some hope that this latter part is adequate, we propose the extensive use of fault tolerance throughout this part of the software.

An overall environment needs to be produced with three clearly defined but cooperating paths for the three development techniques. A monitor is needed which would interact with the user to allow different parts of the requirements specification to be guided down different paths. The requirements specification for some clearly defined part of the software could be presented to an automatic programming system. If the system failed to synthesize the necessary software, the user would then have to write the software and attempt to verify it. If that failed, the user would be required to restructure the software to include the necessary fault tolerance. The monitor would be required to keep track of these various activities and assemble the final program from the various synthesized, verified, and fault tolerant parts.

Figure 8.1 shows this proposed approach in rather limited detail. Essentially, each of the three major aspects of the method is a software development approach in its own right and will be described below in more detail than shown in Figure 8.1.



The Comprehensive Approach

Figure 8.1

8.2. Requirements

It is difficult to choose any one of the various requirements languages that have been developed; each has its advantages and disadvantages. For reliability, extreme formalism is best. The predicate calculus is a good choice because there is such an extensive body of theory supporting it, and it is precise. It is also quite difficult for the average computer scientist to use. For ease of use, natural language is a good choice but there is no supporting formal theory, and natural languages are imprecise and ambiguous. The impressive work of Balzer on the SAFE system leads us to suggest that some form of restricted natural language be used for crucial system requirement specification and processed by the SAFE system to produce formal requirements specifications.

We have not had any direct experience with using the SAFE system. The papers which have been published about the system are rather limited, but the system seems to be very impressive. A major concern is whether it can handle a natural language with sufficient expressive power for crucial systems.

8.3. The Monitor

The monitor is responsible for coordinating all the system activities once the requirements specification has been produced by the SAFE system. It requires a database for maintaining data, source files, reports, etc, as development proceeds. It is not dissimilar from the control systems of existing advanced environments. However, since there are three parallel development paths, the interactions between the paths will have to be handled very carefully. Some of these interactions are touched upon in further subsections. It is probably the case that no existing or currently proposed environment could handle all of these interactions.

8.4. The Automatic Programming System

Which automatic programming system should be used? We cannot select a single method because the technology is so immature. Given the current state of the art however, we suggest the methods of Balzer, and Manna and Waldinger.

There is a conflict between the use of automatic programming systems and the other two major parts of this system. In principle, an automatic programming system is supposed to do everything, including design. The other two parts require human input for everything including the design phase. If the automatic programming system cannot handle the entire development, it may be able to synthesize part of the software. The part which remains may not be in a convenient form for any of the traditional design methods.

Another issue is the difficulty of building automatic programming systems which can handle arbitrary problem domains. The search space that this implies is very large and this is a major limiting factor in the ability of automatic programming systems to synthesize programs.

Both of these problems can be solved in the following way. The automatic programming aspect of this system can be implemented as a series of automatic programming systems operating in parallel and each tackling a small, well-defined part of the crucial software applications domain. For example, most crucial systems operate in real time and an automatic programming system capable of synthesizing real-time schedulers, and nothing else, could be a component of the system. That part of the specification defining the real-time requirements could then be supplied to that module and a suitable scheduler output.

Major aspects of the software design phase would then take place at the monitor stage. Those parts of the software which could be synthesized by the automatic programming systems could be selected and code synthesized. What remained would be well defined and amenable to human detailed design and conventional development. Thus, we propose a set of automatic programming systems, rather than one, until technology reaches the point that a single system can cope with a complete crucial system. As technology proceeds and more powerful automatic programming systems become available, they can be added to such a design, and more of the development of a crucial system can be moved from verification and fault tolerance to automatic programming.

8.5. Verification

Verification is just one part of software development, unlike automatic programming which (in principle) covers the entire transition from requirements specification to executable program. Thus a program which is to be verified requires all the elements of the software development cycle to be present. That is assumed in this comprehensive approach.

The verification part of this system would operate as existing verification systems do. Other parts of the system would be required to assist the process. The monitor and associated database would be used to store proofs, control access to source code, and so on.

A theorem prover would be needed for verification. Many approaches to theorem proving exist and we do not comment on which might be used. However, we note that some automatic programming systems rely on theorem proving. Indeed, the Manna and Waldinger [100] approach derives a program from a proof. Thus a theorem prover is central to both verification and program synthesis, and proof techniques which can be shared by both technologies should be included in this comprehensive approach.

8.6. Conventional Software Development Cycle

The "improved conventional software development cycle" which was delineated in Section 3 is modified from a stand-alone system to fit into the context of our comprehensive approach. The monitor serves in the role of the enhanced programming environment. The SAFE system stands in the role of the requirements specification and analysis stages of the conventional approach (see Figure 3.1). The use of the SAFE system may eliminate the possibility that the requirements specification is incomplete or inconsistent. Gross design has been done by the SAFE system and the monitor. This entails the separation of concerns for the automatic programming systems and the human development effort. The portions of the crucial system developed by the automatic programming systems can (and are intended to) act as components for the human effort. Such portions may also enhance the overall system's prototyping capabilities in that an early prototype may consist almost entirely of the automatically programmable parts of the crucial system. It may occur that the automatic programming systems and verification paths "fail" due to human error in the human-guided gross design. For this reason, the conventional cycle's loop back to original requirements specification has not been eliminated.

8.7. Fault Tolerance

The monitor can require that fault-tolerance be designed into all human programmed software. The code built for attempted verification, if it passes, can serve as is. However, if it fails verification, it can serve as primary alternative in a fault-tolerant design with the monitor then requiring the creation of other alternates. The conditions used in the verification attempt can be saved for use as acceptance tests to fill out the fault-tolerant implementation.

A monitor which "knows" about Safe Programming [114] can aid both in verification and in providing fault-tolerance by enforcing limits on all loops. The monitor should also force the use of fault-tolerant techniques at all interfaces of the human-created code with the outside world and with other parts of the human-created code. All of these interfaces would be known to the monitor since it presided over the high-level design.

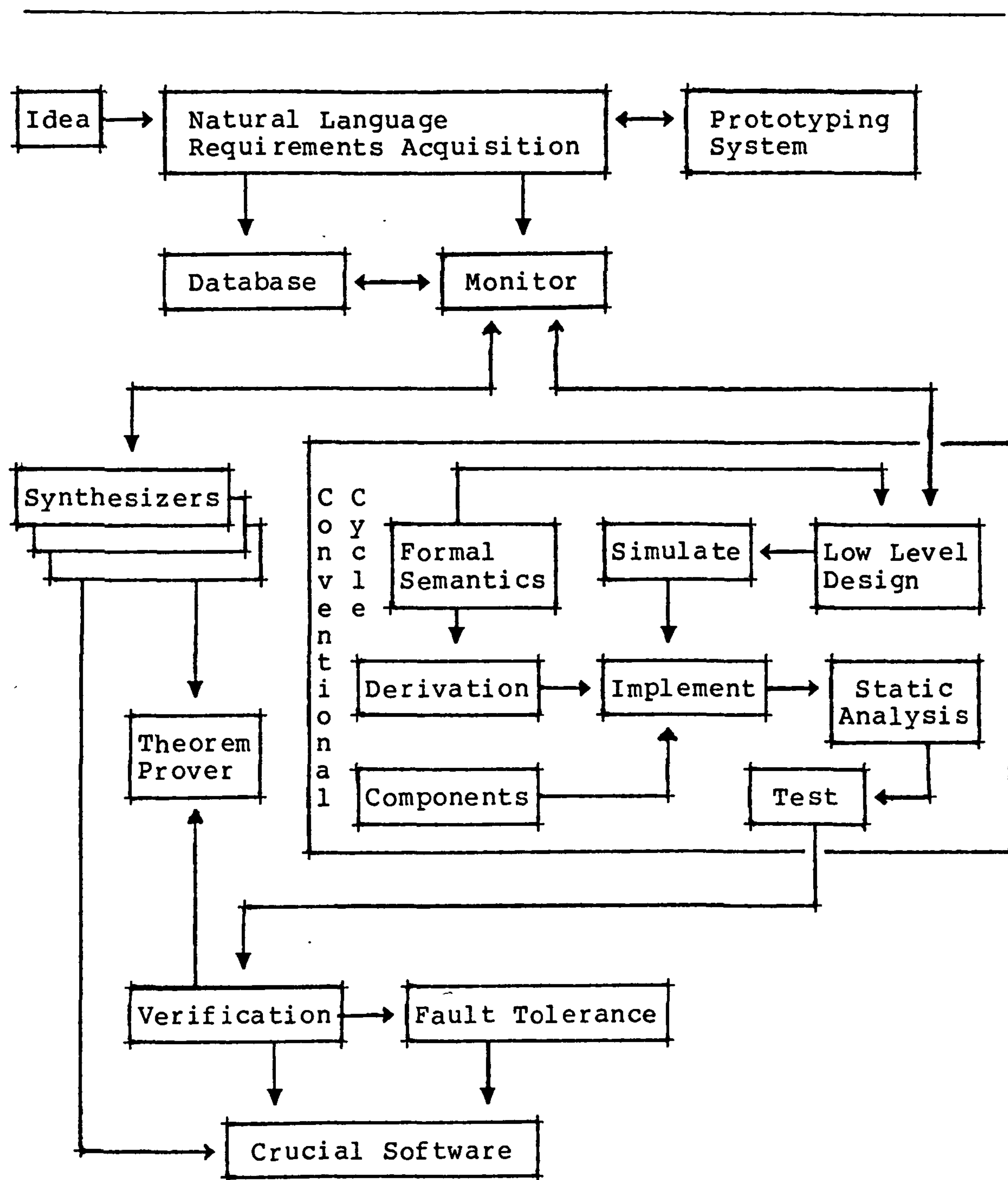
The application of fault-tolerance is not strictly limited to the non-verifiable human effort. The automatically programmed parts need fault-tolerant interfaces with the human's code, both when using a human-programmed component and when being passed parametric information from human-programmed components. One way to create a design fault is for the human to mis-use the automatically programmed code. Although in the strictest sense a requirements specification problem, the automatically programmed portions must be able to handle all possible situations in the critical system/world interface as well.

8.8. Testing

No matter how the software is built, if it is for a crucial application, it will be tested. There are many technologies which are involved in testing. In Section 2 we discussed only commonly-used conventional methods in which the program is executed on sample inputs and the resulting outputs are compared with those expected (recall however that it is frequently difficult to know what output is expected).

Naturally, this approach should be taken with crucial software also. As we have noted however, there is no theoretical basis for any of the practical, conventional testing methods and very little can be concluded about the software from the results of the tests.

Nevertheless, given that testing will occur, how should the test cases be selected and how should the tests be conducted? With no theory of testing, any and all the methods have merit. There is no reason why they cannot all be used. We show in the detailed version of our approach (Figure 8.2) a test case generator which is merely a piece of software designed to aid the programmer in correctly generating the desired combination of inputs. The monitor is shown connected to the testing tools because the tests will be driven by the formal version of the software requirements specification.



The Comprehensive Approach In Detail

Figure 8.2

SECTION 9

AIRLAB Research and Experimentation Recommendations

We use the term AIRLAB here to mean the facility presently being constructed and the research objectives of developing technology to meet the reliability requirements of crucial systems such as digital systems for commercial air transports. We assume that NASA's major interest is in making large improvements in software reliability over the long term via essentially basic research. We also assume that resources are limited and that the most promising technologies need to be selected. Thus these recommendations are limited to those which we consider to be high risk and high payoff. We divide these recommendations into the categories of the enhanced conventional software development cycle, fault tolerance, automatic programming, and the comprehensive approach.

9.1. The Software Development Cycle

There are many areas of research in the software development cycle which could be pursued. It is likely, however, that good progress will be made in some of these areas independently of any NASA sponsored research. For example, there is a great deal of research on environments being funded by the Department of Defense. This work is not oriented particularly to highly-reliable systems, but should yield valuable results and demonstrations which will be of direct benefit to those interested in crucial systems.

In examining the conventional software development cycle and enhancements, suitable topics for research can probably be limited to those areas which are receiving substantially too little attention or where the goals of other researchers do not adequately address the difficulties of crucial software development. The latter is characterized by research motivated by cost reduction, improved programmer productivity, or faster and smaller software. None of these is a very great concern for crucial software where reliability is the dominating metric.

Given these criteria, we suggest the following areas from the enhanced software development cycle be considered for research support:

(1) Rapid Prototyping. The technology is very immature and holds great promise for clarifying issues at the start of a software project.

(2) Requirements Specification. Although there is active research in this area, it is not directed to crucial applications and the state of the art is really very poor.

(3) Software Testing. Despite the amount of testing which is performed and the length of time we have been testing programs, there is

still no scientific basis for testing.

(4) Static Analysis. This is basically an immature technology which seems very promising but still has major problems. It is potentially very valuable in crucial software development because it is automatic.

As well as the above, we suggest that a monitoring project be started to examine and evaluate conventional software development methods. New techniques are continually being developed and reported. How good are they? What is their impact on crucial software development? These questions need to be answered by experts in the development of crucial systems. Many military and commercial crucial systems are presently being built with ad hoc collections of tools, very limited knowledge of the state of the art, and limited resources to follow technology as it evolves. A source of information and assessment of technology as it applies to crucial systems would be very valuable to the developers of these systems. It would also permit a clear assessment of research needs.

9.2. Fault Tolerance

As noted in Section 5 there are many open questions in the technology of fault tolerance. A high priority area of research has to be the resolution of these various issues in order to provide a complete framework for the construction of fault-tolerant crucial systems. Topics include:

(1) Design and construction of hardware to provide processors which include recovery caches and support for the voting necessary in N-version programming.

(2) Determination of a suitable syntax for the conversation technique and its incorporation into a general language structure for fault tolerance based on backward error recovery.

(3) The creation of a better theoretical background for N-version programming and the formulation of a framework which guarantees the atomicity of the versions.

(4) A comprehensive study of the voting issue in N-version programming.

(5) A study of the most appropriate way of combining recovery blocks and N-version programming in the construction of crucial software.

Intuitively, software fault tolerance seems like a good idea. There is precious little evidence, however, showing that it really is. In fact, there is very little evidence showing that software fault tolerance is even feasible. Ideas which seem reasonable in theory sometimes turn out to be impractical, especially in computer science. Some experiments have been done which have implemented fault tolerant systems [] but they were very limited in scope and not in the avionics or even

real-time field.

A major research area to which AIRLAB is ideally suited is the construction of realistic demonstration fault-tolerant systems. We propose that advanced applications such as active controls be taken as typical of the crucial systems which will be required in the near future and that fault-tolerant versions of these applications be constructed. We have no doubt that many significant issues will arise in such activities which have not so far been suggested or resolved.

9.3. Automatic Programming

Automatic programming is very far from practical use but seems to hold great promise. Clearly, it is the highest risk, highest payoff, longest term technology being considered in this report. It has to be understood that the payoff period is likely to be many years [98]. In view of its technical infancy, there are few clear-cut experiments which can be conducted in the AIRLAB framework. Experiments involving program synthesis would probably have to be extremely simple, reflecting the state of the art.

In general, we recommend that automatic programming be reviewed in more depth than has been possible in this study. This review should include detailed evaluation of specific systems by installing them in AIRLAB if possible, and evaluating them carefully in the context of crucial software. The results of these analyses would permit a coordinated research program to be planned. Specifically, we recommend:

(1) A working group of leading researchers in the field be assembled to review the state of the art, compare and contrast systems, and discuss the applicability of the technology to crucial systems.

(2) Install, test and evaluate the SAFE system. Based on published reports, this seems like a very powerful system which could be applied to crucial system requirements specification in the very near future.

(3) Install, test and evaluate the PSI system. Based on published reports (which are extensive), this seems to be the most complete and general automatic programming system that has been built.

(4) Install, test and evaluate any other automatic programming systems which appear to hold any promise of being suitable for programming crucial applications.

Once some experience has been gained with the available automatic programming systems, research goals will become clearer. It may be appropriate to begin constructing an automatic programming system tailored to real-time control although this does not seem desirable at this point. It is important to review existing systems, get the opinions of experts in the field, and gather specimen problems before defining research goals in this area.

9.4. Comprehensive Approach

The comprehensive approach which we proposed in Section 8 attempts to mix several technologies which are not normally used together. A central experiment which we recommend is an attempt to build a version of the comprehensive approach to determine the feasibility of this integration. If this experimental system is built carefully, it should allow new tools, such as more powerful automatic programming systems, to be added and evaluated as they become available. A testbed for new tools or modified environments is essential to allow for the assessment of these technologies in the crucial software context.

As we have noted elsewhere in this report, some software engineering technologies are developed with a specific application area in mind. If this area does not include crucial software, the technology may be useful and it may not. To allow for uniform evaluation, we recommend the establishment of a collection of representative problems from crucial applications. These could be made available to researchers to assist them in evaluating their own work, and they could be used by NASA to evaluate new technologies as they become available.

SECTION 10

Conclusions

We have reviewed many areas of software engineering in an effort to determine which areas of technology could contribute to a major improvement in software reliability if research is pursued vigorously. We have formed the opinion that methods which are presently used for software development are inadequate for building crucial systems. Further, we feel that existing methods are so far from producing the desired level of reliability, and that the required level will not be reached by incremental improvements to commonly used techniques.

As a first step, we propose that the conventional software development cycle be enhanced substantially by integrating the new technologies of software prototypes, software components, fault tolerance, the Ada program language, testing based on the emerging theories of adequate test coverage, and machine-based methodology enforcement. Even using the best modern technology, there seems little hope of achieving the required level of reliability and certainly no hope of being sure that this level has been achieved. The flaws in the conventional development cycle (even if it is substantially enchanced) are the extent to which it relies on human decision making and the non-scientific basis of most of the methodology.

Fault tolerance is often proposed as a "safety net" for software. Supposedly, even if the software contains faults, fault-tolerant methods

will prevent these faults from leading to failures. It may, but this has still to be demonstrated and for concurrent systems (including many real-time systems), fault tolerance still has to be shown to be practical let alone useful.

Formal verification has made remarkable progress and is able to deal with quite sophisticated programs. Unfortunately, there are still major areas where verification is not possible. As a second step therefore we suggest that verification be integrated into the software development cycle and that its use be required wherever possible. Informally, we expect to see systems developed in which those parts of the system amenable to verification are verified and the remainder build by conventional methods. These latter parts would be required to include fault tolerance so that there is some "insurance" against failure in the non-verified parts.

In the long term, really large improvements in reliability will be achieved only if human creativity and decision making are removed from software development. This leads us to suggest that the techniques of automatic programming might provide the source of major reliability improvements. Automatic programming is very limited in its capabilities now but the possibility of direct machine translation of requirements specification to executable program has obvious and major advantages. We propose therefore that automatic programming be pursued as a topic of basic research. It cannot be used in building crucial systems at present but as research advances the state of the art, it could be used to build gradually larger parts of crucial systems.

Our comprehensive approach is a combination of automatic programming, verification, and fault tolerance coupled to improved conventional methods. The approach involves a system in which all three paths are available. A crucial system would be constructed by synthesizing as much as possible (which may be very little) using an automatic programming system, building the remainder using conventional methods and verifying as much as possible, and finally employing fault tolerance for those parts which cannot be synthesized or verified.

This approach will not necessarily improve reliability, but, even if it does, it may be very difficult to ensure that desired levels of reliability have been achieved. However, as basic research on automatic programming and verification allow more of a crucial system to be built with these technologies, reliability will surely increase. When systems can finally be totally synthesized automatically, it may be possible to make definitive statements about reliability.

REFERENCES

The works below are referenced within this report. Most also occur within the subsequent bibliographies.

1. K. L. Heninger, "Specifying Software Requirements for Complex Systems: New Techniques and Their Applications," _IEEE Transactions on Software Engineering_, Vol. 6(1), pp. 2-12 (Jan 1980).

2. T. Bell, D. Bixler, and M. Dyer, "An Extendable Approach to Computer-Aided Software Requirements Engineering," _Software Engineering Technology; State of the Art Report_, Infotech International.

3. S. H. Caine and E. K. Gordon, "PDL -- A Tool for Software Design," pp. 380-385, in _Tutorial on Software Design Techniques_, IEEE Computer Society (1980).

4. S. A. Sutton and V. R. Basili, "The FLEX Software Design System: Designers Need Languages, Too," _IEEE Computer_, Vol. 14(11), p. 95 (Nov. 1981).

5. D. Teichroew and E. A. Hershey, "PSL/PSA: A Computer-aided Technique for Structured Documentation and Analysis of Information Processing Systems," _IEEE Transactions on Software Engineering_, Vol. 3(1), p. 41 (1977).

6. (author n.a.), in Reference Manual for the Ada Programming Language, United States Department of Defense (1981).

7. W. M. Turski, "Software Engineering: Some Principles and Problems," pp. 485-491, in Mathematical Structures, Computational Mathematics, Mathematical Modelling, Bulgarian Academy of Sciences, Sofia (1975).

8. R. J. Lauber, "Development Support Systems," IEEE Computer, Vol. 15(5), p. 36 (May 1982).

9. K. L. Heninger, J. Kallander, D. L. Parnas, and J. Shore, "Software Requirements for the A-7E Aircraft," Memo Rep. 1876, Naval Research Lab., Washington, D.C. (Nov. 27, 1978).

10. M. Alford, "SREM," Proceedings 3rd Digital Avionics Systems Conference, IEEE (November 1979).

11. M. Alford, "SREM at the Age of Two," Proceedings COMPSAC (November 1978).

12. R. Loshbough, M. Alford, J. T. Lawson, D. M. Sims, and T. R. Johnson, "Applicability of SREM to the Verification of MIS System Software Requirements (Vol I and II)," NTIS reference AD-A100 720/2 Volume I ; AD-A120 721/0 Volume II.

13. (author n.a.), in BMDATC Software Development System, Ballistic Missile Defense Advanced Technology Center (July 1975).

14. N. Wirth, "Program Development by Stepwise Refinement," Communications of the ACM, Vol. 14(4), p. 221 (Apr. 1971).

15. F. P. Brooks, in The Mythical Man Month, Addison-Wesley (1975).

16. G. D. Bergland, "A Guided Tour of Program Design Methodologies," IEEE Computer, Vol. 14(10), p. 13 (Oct. 1981).

17. M. A. Jackson, in Principles of Program Design, Academic Press, London (1975).

18. E. W. Dijkstra, in A Discipline of Programming, Prentice-Hall (1976).

19. R. W. Floyd, "Assigning Meaning to Programs," Proceedings of the Symposium in Applied Mathematics, pp. 19-32, American Mathematical Society (1967).

20. C. A. R. Hoare, "An Axiomatic Basis for Computer Programming," pp. 89, in Programming Methodology, ed. D. Gries, Springer-Verlag, New York (1978).

21. K. R. Apt, "Ten Years of Hoare's Logic: A Survey - Part I," ACM Transactions on Programming Languages and Systems, Vol. 3(4), p. 431 (October 1981).

22. D. L. Parnas, "On the Criteria to be Used in Decomposing Systems into Modules," Communications of the ACM (Dec. 1972).

23. D. L. Parnas, "On a 'Buzzword': Hierarchical Structure," pp. 336, in IFIP 1974 (1974).

24. G. Booch, "Object Oriented Design," Ada Letters, Vol. I(3), p. 64 (Mar.-Apr. 1982).

25. G. J. Myers, in _Reliable Software Through Composite Design_, Petrocelli/Charter (1975).

26. W. P. Stephens, G. J. Myers, and L. L. Constantine, "Structured Design," _IBM Systems Journal_, Vol. 13(2), p. 115 (1974).

27. E. Yourdon and L. Constantine, in _Structured Design_, Prentice-Hall (1979).

28. V. Basili and A. J. Turner, "Iterative Enhancement: A Practical Technique for Software Development," _IEEE Transactions on Software Engineering_, Vol. 1(4), p. 390 (Dec. 1975).

29. T. Gilb, "Evolutionary Development," _Software Engineering Notes_, Vol. 6(2), p. 17 (Apr. 1981).

30. L. Robinson, "The HDM Handbook," 4828, Computer Science Group SRI International (1979).

31. R. S. Boyer and J. S. Moore, "Proving Theorems About LISP Programs," _Journal of the ACM_, Vol. 22(1), p. 48 (1975).

32. N. Wirth, "Toward a Discipline of Real-Time Programming," _CACM_, Vol. 20(8), pp. 577-583 (Aug 1977).

33. J. Holden and I. C. Wand, "An Assessment of Modula," _Software-Practice & Experience_, Vol. 10(8), pp. 593-621 (Aug. 1980).

34. N. Wirth, "Modula Language Tools," _Presentation for Washington D.C. Chapter ACM at U. of Maryland_ (3/18/82).

35. F. H. Martin, _HAL/S - The Avionics Programming System for Shuttle_.

36. J. R. Garman, "The 'Bug' Heard 'Round the World," Software Engineering Notes, Vol. 6(5), pp. 3-10 (Oct 1981).

37. W. R. Collins, S. T. Gregory, and R. Hamm, in Supporting the execution of HAL/S Programs at NASA LaRC (April 1981).

38. M. MacLaren, "Evolving Toward Ada in Real Time Systems," Sigplan Notices, Vol. 15(11), p. 146 (Nov. 1980).

39. A. R. Tripathi, W. D. Young, D. I. Good, and J. C. Browne, "HAL/S/V: A Verifiable Subset for HAL/S," Sigplan Notices, Vol. 16(3), p. 102 (Mar 1981).

40. J. Stroet, "An Alternative to the Communication Primitives in Ada," SIGPLAN Notices, Vol. 15(12), pp. 62-74 (Dec. 1980).

41. J. C. Knight, slides from presentation for Hampton chapter of ACM.

42. R. T. Boute, "Simplifying Ada by Removing Limitations," Sigplan Notices, Vol. 15(2), p. 17 (Feb 1980).

43. E. S. Roberts, A. Evans, C. R. Morgan, and E. M. Clarke, "Task Management in Ada- A Critical Evaluation for Real-time Multiprocessors," Software-Practice & Experience, Vol. 11(10), pp. 1019-1052 (Oct. 1981).

44. A. Mahjoub, "Some Comments on Ada as a Real-Time Programming Language," Sigplan Notices, Vol. 16(2), pp. 89-95 (Feb 1981).

45. A. Silberschatz, "On The Synchronization Mechanism of the Ada Language," SIGPLAN Notices, Vol. 16(2), pp. 96-103 (Feb. 1981).

46. D. W. Jones, "Tasking and Parameters: A Problem Area in Ada," Sigplan Notices, Vol. 15(5), pp. 37-40 (May 1980).

47. S. Haridi, J. Bauner, and G. Svensson, "An Implementation and Empirical Evaluation of the Tasking Facilities in Ada-Summary," Sigplan Notices, Vol. 16(2), pp. 35-47 (Feb 1981).

48. D. A. Bennett, B. D. Kornman, and J. R. Wilson, "Hidden Costs in Ada," Ada Letters, Vol. I(4), pp. I-4.9 (May-June 1982).

49. B. Brosgol, "Summary of Language Changes," Ada Letters, Vol. I(3), p. 34 (Mar-Apr 1982).

50. W. H. Jessop, "Ada Packages and Distributed Systems," Sigplan Notices, Vol. 17(2), pp. 28-36 (Feb 1982).

51. J. C. D. Nissen, P. Wallis, and B. A. Wichmann, "Ada-Europe Guidelines for the Portability of Ada Programs," Ada Letters, Vol. I(3), pp. 44-61 (Mar-Apr 1982).

52. W. Eventoff, D. Harvey, and R. J. Price, "The Rendezvous and Monitor Concepts: Is there an Efficiency Difference?," Sigplan Notices, Vol. 15(11), pp. 156-165 (Nov 1980).

53. J. B. Goodenough and S. L. Gerhart, "Toward a Theory of Test Data Selection," IEEE Transactions on Software Engineering, Vol. SE-1(2), p. 156 (1975).

54. M. Geller, "Test Data as an Aid in Proving Program Correctness," Communications of the ACM, Vol. 21(5), p. 368 (May 1978).

55. E. H. Forman and N. D. Singpurwalla, "An Empirical Stopping Rule for Debugging and Testing Computer Software," Journal of the American Statistical Association, Vol. 72, pp. 750-757 (Dec. 1977).

56. Musa, "The Measurement and Management of Software Reliability," Proceedings of the IEEE, Vol. 68(9) (Sept. 1980).

57. Littlewood, "Theories of Software Reliability: How Good Are They and How Can They Be Improved?," Transactions on Software Engineering, Vol. SE-6(5), p. 489 (Sept. 1980).

58. L. A. Clarke, J. Hassell, and D. J. Richardson, "A Close Look at Domain Testing," IEEE Transactions on Software Engineering, Vol. 8(4), p. 380 (Jul. 1982).

59. D. Prentice, "An Analysis of Software Development Environments," Software Engineering Notes, Vol. 6(5), p. 19 (Oct. 1981).

60. T. A. Dolotta, R. C. Haight, and J. R. Mashey, "The Programmer's Workbench," Bell System Technical Journal (Jul.-Aug. 1978).

61. T. A. Dolotta and J. R. Mashey, "An Introduction to the Programmer's Workbench," pp. 164, in Proceedings of the Second International Conference on Software Engineering (1976).

62. R. E. Millstein, "The National Software Works: A Distributed Processing System," pp. 44, in Proceedings of the ACM 77 Annual Conference (Sept. 1977).

63. (author n.a.), in Requirements for Ada Programming Support Environments: STONEMAN, United States Department of Defense (Feb. 1980).

64. D. A. Fisher, Design Issues for Ada Program Support Environments: A Catalogue of Issues, Science Applications, Inc., McLean, Va. (Oct. 1980).

65. M. A. Branstad, W. R. Adrion, W. Howden, L. Osterweil, T. Standish, and M. Zelkowitz, "NBS Workshop Report on Programming Environments," Software Engineering Notes, Vol. 6(4), pp. 1-51 (Aug. 1981).

66. W. Teitelman, "The INTERLISP Programming Environment," IEEE Computer, Vol. 14(4), p. 25 (Apr. 1981).

67. E. Sandewall, "Programming in the Interactive Environment: The LISP Experience," ACM Computing Surveys, Vol. 10(1), p. 35 (Mar. 1978).

68. P. J. Denning, "'Smart Editors', ACM President's Letter," Communications of the ACM, Vol. 24(8), p. 491 (Aug. 1981).

69. E. Shapiro, G. Collins, L. Johnson, and J. Ruttenberg, "PASES: A Programming Environment for Pascal," SIGPLAN Notices, Vol. 16(8), p. 50 (Aug. 1981).

70. T. Teitelbaum and T. Reps, "The Cornell Program Synthesizer: A Syntax-Directed Programming Environment," Communications of the ACM, Vol. 24(9), p. 563 (Sept. 1981).

71. S. B. Whitehall, "An Ada Virtual Operating System," Proceedings of the AdaTEC Conference on Ada (ACM Order #825821), p. 238 (Oct. 1982).

72. Y. Mano, K. Omaki, and K. Torii, "An Intelligent Multi-display Terminal System -- Towards a Better Programming Environment," Software Engineering Notes, Vol. 6(2), p. 8 (Apr. 1981).

73. (author n.a.), "UNIX Issue," Bell System Technical Journal (Jul.-Aug. 1978).

74. B. W. Kernighan and J. R. Mashey, "The UNIX Programming Environment," IEEE Computer, Vol. 14(4), p. 12 (Apr. 1981).

75. W. Teitelman, A Display-Oriented Programmer's Assistant, Xerox PARC (Mar. 1977).

76. J. Ramanathan and C. J. Shubra, "Modeling of Problem Domains for Driving Program Development Systems," pp. 28, in Proceedings of the Eighth Annual Symposium on Principles of Programming Languages (Jan. 1981).

77. P. M. Cashman and A. W. Holt, "A Communications Oriented Approach to Structuring the Software Maintenance Environment," Software Engineering Notes, Vol. 5(1), p. 4 (Jan. 1980).

78. D. A. Smith, "Rapid Software Prototyping," 187, Department of Information and Computer Science, University of California, Irvine (May 1982).

79. S. L. Squires, M. Branstad, and M. Zelkowitz, "Special Issue on Rapid Prototyping: Working Papers from the ACM Rapid Prototyping Workshop," Software Engineering Notes, Vol. 7(5), pp. 1-185 ACM Order Number 592831 (Dec. 1982).

80. J. M. Neighbors, "Software Construction Using Components," 160, Department of Information and Computer Science, University of California, Irvine (May 1980).

81. T. W. Pratt, "H-Graph Semantics," 81-15, 81-16, DAMACS University of Virginia (Sept. 1981).

82. T. W. Pratt, "Formal Specification of Software Using H-Graph Semantics," 83-2, DAMACS University of Virginia (Jan. 1983).

83. A. Dean, in _personal communication_.

84. P. M. Melliar-Smith and R. L. Schwartz, "Formal Specification and Mechanical Verification of SIFT: A Fault-Tolerant Flight Control System," _IEEE Transactions on Computers_, Vol. C-31(7), p. 616 (July 1982).

85. J. Goguen and K. Levitt, "Experiences and Perspectives with SRI's Tools for Software Design and Validation," _7th Annual Software Engineering Conference_ --_GSFC_ (Dec. 1982).

86. J. M. Nye, in _Who, What, & Where in Communications Security_, Marketing Consultants International, Inc., Hagerstown, Maryland (Apr. 1981).

87. J. R. Dunham and J. C. Knight, "Production of Reliable Flight-Crucial Software," NASA Conference Publication 2222 (1982).

88. T. Winograd, "Beyond Programming Languages," _Communications of the ACM_, Vol. 22(7), p. 391 (July 1979).

89. A. I. Wassermann, "Software Engineering: The Turning Point," *IEEE Computer*, Vol. 11(9), p. 30 (Sept. 1978).

90. K. H. Kim, "Approaches to Mechanization of the Conversation Scheme Based on Monitors," *IEEE Transactions on Software Engineering*, Vol. SE-8(3), pp. 189-197 (May 1982).

91. D. L. Russel and M. J. Teideman, "Multiprocess Recovery Using Conversations," *Digest of Papers FTCS-9*, p. 106 (June 1979).

92. T. Anderson and J. C. Knight, "A Framework for Fault-Tolerance in Real-Time Systems," *IEEE Transactions on Software Engineering* (Apr. 1983).

93. L. Chen and A. Avizienis, "N-Version Programming: A Fault-Tolerance Approach to Reliability of Software Operation," *Digest of Papers FTCS-8*, p. 3 (June 1978).

94. D. J. Martin, "Dissimilar Software in High-Integrity Applications in Flight Controls," pp. 36-1, in *AGARD Conference on Software for Avionics* (Sept. 1982).

95. T. Anderson, in *personal communication*.

96. R. S. Boyer, M. W. Green, and J. S. Moore, "The Use of a Formal Simulator to Verify a Simple Real Time Control Program," ICSCA-CMP-29, Institute for Computing Science, The University of Texas at Austin (July 1982).

97. D. Good, in *personal communication*.

98. A. W. Biermann, "Approaches to Automatic Programming," _Advances in Computers_, Vol. 15, pp. 1-63 (1976).

99. A. Barr and E. A. Feigenbaum, in _The Handbook of Artificial Intelligence_, William Kaufmann, Inc., Los Altos, Ca. (1982).

100. Z. Manna and R. J. Waldinger, "Towards Automatic Program Synthesis," _Communications of the ACM_, Vol. 14(3), pp. 151-164 (1971).

101. R. Balzer and N. Goldman, "Informality in Program Specifications," _IEEE Transactions on Software Engineering_, Vol. SE-4(2), pp. 94-103 (March 1978).

102. M. M. Zloof, "Query by Example: a Database Language," _IBM Systems Journal_, Vol. 16(4), p. 324 (1977).

103. M. M. Zloof, "Office by Example: a Business Language that Unifies Data and Word Processing and electronic Mail," _IBM Systems Journal_, Vol. 21(3), p. 272 (1982).

104. Z. Manna and R. Waldinger, "A Deductive Approach to Program Synthesis," _ACM Transactions on Programming Languages and Systems_, Vol. 2(1), pp. 90-121 (January 1980).

105. R. Balzer, "Transformational Implementation: An Example," _IEEE Transactions on Software Engineering_, Vol. SE-7(1), pp. 3-14 (January 1981).

106. R. M. Balzer, in _personal communication_ (1983).

107. C. Rich, "A Library of Programming Plans with Applications to Automatic Analysis, Synthesis and verification of Programs," Doctoral Dissertation, Massachusetts Institute of Technology (1979).

108. C. Rich and H. E. Shrobe, "Initial Report on a LISP Programmer's Apprentice," IEEE Transactions on Software Engineering, Vol. SE-4(6), pp. 456-467 (1978).

109. H. E. Shrobe, "Reasoning and Logic for Complex Program Understanding," Doctoral Dissertation, Massachusetts Institute of Technology (1978).

110. R. C. Waters, "Automatic Analysis of the Logical Structure of Programs," AI-TR-492, Massachusetts Institute of Technology (1978).

111. C. Green, "The Design of the PSI Program Synthesis System," Proceedings of the 2nd International Conference on Software Engineering, pp. 4-18.

112. D. R. Barstow, in Knowledge-Based Program Construction, Elsevier North-Holland (1979).

113. E. Kant and D. R. Barstow, "The Refinement Paradigm: The Interaction of Coding and Efficiency Knowledge in Program Synthesis," IEEE Transactions on Software Engineering, Vol. SE-7(5), p. 458 (Sept. 1981).

114. T. Anderson and R. W. Witty, "Safe Programming," BIT, Vol. 18(1), pp. 1-8 (1978).

INTRODUCTION TO THE BIBLIOGRAPHIES

The following extended bibliographies are intended to provide the reader with starting points for exploring further any of the subjects addressed in this report. These bibliographies list books, articles, reports, and papers to be found in the literature on the subjects discussed herein. There is some minor duplication among the sections since authors often deal with more than one subject within a work. Further, some of the more obscure works were not actually located, but the references obtained from the references sections of works which were located. Hence, there may be some inherited inaccuracies. The topics covered are:

(1) Requirements Engineering.

(2) Distributed Specifications.

(3) Design Methodologies.

(4) Parallel Programming Languages.

(5) Testing Methodologies.

(6) Static Analysis.

(7) Symbolic Execution.

(8) Programming Environments.

(9) Software Prototyping.

(10) Functional Languages.

(11) Verification.

(12) Fault Tolerance.

(13) Automatic Programming.

BIBLIOGRAPHY ON REQUIREMENTS ENGINEERING

1. M. Alford, "A Requirements Engineering Methodology for Real-Time Processing Requirements," IEEE Transactions on Software Engineering, Vol. 3(1), pp. 60-69 (Jan 1977).

2. M. Alford, "A Requirements Engineering Technology for Real-Time Processing Requirements," Software Engineering Technology; State of the Art Report, Infotech International.

3. M. Alford, "SREM," Proceedings 3rd Digital Avionics Systems Conference, IEEE (November 1979).

4. M. Alford, "SREM at the Age of Two," Proceedings COMPSAC (November 1978).

5. M. W. Alford and I. F. Burns, "An Approach to Stating Real-Time Software Requirements," Conference on Petri Nets and Regulated Methods, MIT (July, 1975).

6. M. W. Alford and I. F. Burns, "R-nets: A Graph Model for Real-Time Software Requirements," MRI Conference on Software Engineering (April 1976).

7. M. W. Alford, "A Requirements Engineering Methodology for Real-Time Processing Requirements," Proceedings 2nd International Conference on Software Engineering, San Francisco, CA (1976).

8. R. Balzer and N. Goldman, "Informality in Program Specifications," IEEE Transactions on Software Engineering, Vol. SE-4(2), pp. 94-103 (March 1978).

9. R. Balzer and N. Goldmann, "Principles of Good Software Specification and Their Implications for Specification Languages," pp. 58-67, in Proceedings IEEE Conference on Specifications of Reliable Software (1979).

10. V. R. Basili and D. M. Weiss, "Evaluation of a Software Requirements Document by Analysis of Change Data," Proceedings 5th International Conf on Software Engineering.

11. T. Bell, D. Bixler, and M. Dyer, "An Extendable Approach to Computer-Aided Software Requirements Engineering," Software Engineering Technology; State of the Art Report, Infotech International.

12. T. E. Bell and D. C. Bixler, "A Flow-Oriented Requirements Statement Language," MRI Conference on Software Engineering (April 1976).

13. T. E. Bell and T. A. Thayer, "Software Requirements: Are They Really a Problem?," Proceedings 2nd International Conference on Software Engineering (1976).

14. W. Bezanson, "Reliable Software thru Requirements Definition Using Data Abstractions," Microelectronics and Reliability (GB), Vol. 17(1) (January 1978).

15. J. Biewald, E. Joho, and H. Schelling, "Real-Time Features of EPOS: Formulation, Evaluation, and Documentation," in _Proceedings IFAC/IFIP Workshop on Real-Time Programming_, ed. V. H. Haase, Pergamon Press, Oxford-New York (1981).

16. J. Biewald, E. Joho, S. Jovalekic, and H. Schelling, "Application of the Specification and Design Technique EPOS to a Process Control Problem," pp. 517-522, in _Proceedings IFAC Symposium on Digital Computer Applications to Process Control_, ed. R. Isermann & H. Kaltenecker, Pergamon Press, Oxford-New York (1980).

17. J. Biewald, "EPOS -- A Specification and Design Technique for Computer Controlled Real-Time Automation Systems," pp. 245-250, in _Proceedings Fourth International Conference on Software Engineering_, IEEE Computer Society (1979).

18. S. H. Caine and E. K. Gordon, "PDL -- A Tool for Software Design," pp. 380-385, in _Tutorial on Software Design Techniques_, IEEE Computer Society (1980).

19. S. H. Caine and E. K. Gordon, "PDL -- A Tool for Software Design," _AFIPS Conference Proceedings_, Vol. 44, pp. 271-276 (1975).

20. C. C. Carpenter and L. L. Tripp, "Software Design and Validation Tool," _Proceedings 1975 Conference on Reliable Software_ (April 1975).

21. B. E. Casey and B. J. Taylor, "Writing Requirements in English: A Natural Alternative," pp. 95-101, in _IEEE Workshop on Software Engineering Standards_ (Aug. 1981).

22. J. Darlington, "Programming Transformation: An Introduction and Survey," Computing Bulletin, Vol. 2(22), pp. 22-24 (Dec. 1979).

23. B. Dasarathy, "Test Plan Generation for the Requirements Validation of Real-Time Systems," in IEEE Workshop on Automatic Test Program Generation (Apr. 1981).

24. B. Dasarathy, "Timing Constraints for Real-Time Systems: constructs for Expressing Them, Methods of Validating Them," 80-462.6, GTE Laboratories, Waltham, Mass. (Dec. 1980).

25. A. Davis, T. Miller, E. Rhode, and B. Taylor, "RLP: An Automated Tool for Processing of Requirements," GTE Automation and Electronics Journal, Vol. 17(6) (Nov. 1979).

26. A. Davis and T. Rauscher, "A Survey of Techniques Used for Requirements Definition of Computer-based Systems," GTE Automation and Electronics Journal, Vol. 17(5) (Sept. 1979).

27. A. M. Davis and T. G. Rauscher, "Formal Techniques and Automatic Processing to Ensure Correctness in Requirements Specification," pp. 15-35, in Proceedings Specifications of Reliable Software Conference (1979).

28. A. M. Davis, "Automating the Requirements Phase: Benefits to Later Phases of the Software Life-Cycle," pp. 42-48, in Proceedings Compsac 80, IEEE Computer Society (Oct. 1980).

29. A. M. Davis, "Rapid Prototyping Using Executable Requirements Specifications," in ACM Workshop on Rapid Prototyping (Apr. 1982).

30. A. M. Davis, "RLP: An Automated Tool for the Automatic Processing of Requirements," pp. 188-194, in _Proceedings Compsac 79_, IEEE Computer Society (1979).

31. A. M. Davis, "The Design of a Family of Application-Oriented Requirements Languages," _IEEE Computer_, Vol. 15(5), p. 21 (May 1982).

32. J. Demetrovics, E. Knuth, and P. Rado, "Specification Meta Systems," _IEEE Computer_, Vol. 15(5), p. 29 (May 1982).

33. C. R. Everhart, _User Experience with a Formally Defined Requirements Language IORL (Input/Output Requirements Language)_, Teledyne Brown Engineering, Huntsville, Ala. (1978).

34. V. H. Haase and G. R. Koch, "Application-Oriented Specifications," _IEEE Computer_, Vol. 15(5), p. 10 (May 1982).

35. M. Hammer and G. Ruth, "Automating the Software Development Process," in _Research Directions in Software Technology_, ed. P. Wegner, MIT Press (1979).

36. H. C. Heacox, "RDL -- A Language for Software Development," _Sigplan Notices_, Vol. 14(12), pp. 71-79 (Dec. 1979).

37. K. L. Heninger, J. Kallander, D. L. Parnas, and J. Shore, "Software Requirements for the A-7E Aircraft," Memo Rep. 1876, Naval Research Lab., Washington, D.C. (Nov. 27, 1978).

38. K. L. Heninger, "Specifying Software Requirements for Complex Systems: New Techniques and Their Applications," _IEEE Transactions on Software Engineering_, Vol. 6(1), pp. 2-12 (Jan 1980).

39. D. L. Huebner, "Systems Validation Through Automated Requirements Verification," Proceedings COMPSAC (1979).

40. J. Jones, "Bibliography, European Workshop on Industrial Computer Systems," TC 11, Hatfield Polytechnic, Hatfield, Herts. England.

41. G. Kampen, "Experience Using Automated Software Tools in the Specification and Design of a Large Aerospace Application," in Proceedings Conference Use of Computer Aids in Systems Development, Systems Designers Limited, UK (1981).

42. K. C. Kang, "A Methodology for Analyzing System Descriptions for Completeness," in Ph.D. Dissertation, University of Michigan (1980).

43. L. C. Klos, "Software Engineering Systems Requirements," Proceedings 3rd Digital Avionics Systems Conference, IEEE (November 1979).

44. N. Komoda, "An Innovative Approach to Systems Requirement Analysis by Using Structural Modeling Methods," Proceedings 5th International Conference on Software Engineering.

45. J. Kramer, G. Koch, J. Ludewig, M. Dehnert, and D. Vojnovic, "Glossary of Terms of the EWICSTC Application Oriented Specification," Working Paper EWICS (1981).

46. K. Krause and L. Diamant, "A Management Methodology for Testing Software Requirements," Proceedings COMPSAC (November 1978).

47. S. Lamb, V. Leck, L. Peters, and G. Smith, "SAMM: A Modeling Tool for Requirements and Design Specifications," Proceedings

COMPSAC (November 1978).

48. R. J. Lauber, "Development Support Systems," IEEE Computer, Vol. 15(5), p. 36 (May 1982).

49. A. A. Levene and G. P. Mullery, "An Investigation of Requirements Specification Languages: Theory and Practice," IEEE Computer, Vol. 15(5), p. 50 (May 1982).

50. B. Liskov and V. Berzins, "An Appraisal of Program Specifications," pp. 13.1-13.24, in Proceedings Conference on Research Directions in Software Technology (Oct. 1977).

51. B. Liskov and S. Zilles, "Specification Techniques for Data Abstractions," IEEE Transactions on Software Engineering, Vol. 1, pp. 7-19 (Mar. 1975).

52. R. Loshbough, M. Alford, J. T. Lawson, D. M. Sims, and T. R. Johnson, "Applicability of SREM to the Verification of MIS System Software Requirements (Vol I and II)," NTIS reference AD-A100 720/2 Volume I ; AD-A120 721/0 Volume II.

53. J. Ludewig and W. Streng, "Methods and Tools for Software Specification and Design -- A Survey," EWICS TC 7 Paper 149, Zurich (Apr. 1978).

54. J. Ludewig, "Computer-Aided Specification of Process Control Systems," IEEE Computer, Vol. 15(5), p. 12 (May 1982).

55. J. Ludewig, "PCSL -- A Process Control Software Specification Language," in Proceedings IFAC/IFIP Workshop on Real-Time Programming, ed. V. H. Haase, Pergamon Press, Oxford-New York (Apr.

1980).

56. T. J. Miller and B. J. Taylor, "A System Requirements Methodology," pp. 18.5.1-18.5.5, in _IEEE Electro 81 Conference Proceedings_ (Apr 1981).

57. D. L. Parnas and G. Handzel, _More on Specification Teechniques for Software Modules,_ Fachbereich Informatik, Technische Hochschule Darmstadt, FDR (1975).

58. D. L. Parnas and H. Wurges, "Response to Undesired Events in Software Systems," pp. 437-446, in _Procedings 2nd International Conference on Software Engineering_ (1976).

59. D. L. Parnas, "Designing Software for Ease of Extension and Contraction," in _Proceedings 3rd International Conference on Software Engineering_ (May 1978).

60. D. L. Parnas, "Desired System Behavior in Undesired Situations," in _Software Engineering Principles_ (_Document UE.1_), Naval Research Lab., Washington, D.C. (1978).

61. D. L. Parnas, "The Use of Precise Specifications in the Development of Software," in _Proceedings IFIP Congress_ (1977).

62. D. L. Parnas, "Use of Abstract Interfaces in the Development of Software for Embedded Computer Systems," 8047, Naval Research Lab., Washington, D.C. (1977).

63. M. Alford and D. Parnas, "A Technique for Software Module Specification with Examples," _Communications of the ACM,_ Vol. 15(5), pp. 330-336 (May 1972).

64. D. L. Parnas, "The Use of Precise Specifications in the Development of Software," pp. 861-867, in Information Processing 77, ed. B. Gilchrist, North Holland Publishing Company, Amsterdam (1977).

65. G. D. Pratten and R. A. Snowden, "CADES -- Support for the Development of Complex Software," in Proceedings European Computer Conference on Software Engineering, Online, Uxbridge, England (1975).

66. C. Ramamoorthy, "A Survey of Principles and Technology of Software Requirements Specifications," Software Engineering Technology; State of the Art Report, Infotech International.

67. J. G. Rice, Build Program Technique -- A Practical Approach for the Development of Automatic Software Generation Systems, John Wiley, New York (1981).

68. W. E. Riddle, "DREAM -- A Software Design Aid System," in Third Jerusalem Conference Information Processing Technology, Elsevier-North Holland (1978).

69. L. Robinson and O. Roubine, "SPECIAL -- A Specification and Assertion Language," SRI Technical Report CSL-46 (Jan 1977).

70. D. T. Ross and K. E. Schoman, "Structured Analysis for Requirements Definition," IEEE Transactions on Software Engineering, Vol. 3(1), pp. 6-15 (Jan. 1977).

71. D. T. Ross and K. E. Schoman, Jr., "Structured Analysis for Requirements Definition," Proceedings 2nd International Conference on Software Engineering (1976).

72. W. W. Royce, "Software Requirements Analysis, Sizing, and Costing," _Practical Strategies for Developing Large Software Systems,_ Addison-Wesley (1975).

73. S. A. Stephens and L. L. Tripp, "Requirements Expression and Verification Aid," _Proceedings 3rd International Conference on Software Engineering_ (May 1978).

74. B. J. Taylor, "A Method for Expressing the Functional Requirements of Real-Time Systems," pp. 111-120, in _IFAC/IFIP Workshop on Real-Time Programming_ (Apr. 1980).

75. D. Teichroew and E. A. Hershey, "PSL/PSA: A Computer-Aided Technique for Structured Documentation and Analysis of Information Processing Systems," _IEEE Transactions on Software Engineering,_ Vol. 3(1), pp. 60-69 (Jan 1977).

76. A. O. Ward, "A Consistent Approach to the Development of System Requirements and Software Design," in _Proceedings Conference Use of Computer Aids in Systems Development,_ Systems Designers Limited, UK (1981).

77. A. Wasserman and S. Stinson, "A Specification Method for Interactive Information Systems," _Proceedings Conference on Reliable Software_ (April 1979).

78. R. R. Willis, "AIDES -- Computer Aided Design of Software Systems," in _Proceedings Symposium Software Engineering Environments,_ Elsevier-North Holland (1980).

79. E. W. Winters, "Practical Experience with the Problem Statement Language," in _Proceedings Conference Use of Computer Aids in Systems Development_, Systems Designers Limited, UK (1981).

80. R. T. Yeh and P. Zave, "Specifying Software Requirements," _Proceedings of the IEEE_, Vol. 68, pp. 1077-1085 (1980).

81. P. Zave and R. T. Yeh, "Executable Requirements for Embedded Systems," _Proceedings 5th International Conference on Software Engineering_.

82. "Cascade/2 in Brief," in _Runit_, University of Trondheim, Norway.

83. _Handbook of the Pilot SDS_, Ministry of Defense R.S.R.E. Malvern, UK.

84. _Requirement Specification Language Study REP7: Specification of Requirements Using an Automated Aid_, Systems Designers Limited.

85. _Requirement Specification Language Study WP5: A Survey of Automated Techniques for System Description_, System Designers Limited.

86. _The Official Handbook of MASCOT_, Mascot Suppliers Association, London (1980).

BIBLIOGRAPHY ON DISTRIBUTED SPECIFICATIONS

1. J. Bacon, "An Approach to Distributed Software Systems," Operating Systems Review, Vol. 15(4), pp. 62-74 (October 1981).

2. M. Ben-Ari, "Cheap Concurrent Programming," Software--Practice and Experience, Vol. 11(12), pp. 1261-1264 (December 1981).

3. R. Balzer and N. Goldman, "Principles of Good Software Specification and Their Implications for Specification Languages," AFIPS Conference Proceedings, 1981 NCC, Vol. 50, pp. 393-400 (May 4-7, 1981).

4. K. Hirose, "Specification Technique for Parallel Processing: Process-Data Representation," AFIPS Conference Proceedings, 1981 NCC, Vol. 50, pp. 409-413 (May 4-7, 1981).

5. G. M. Booth, Distributed Systems Environment: Some Practical Approaches, McGraw-Hill, New York (1981).

6. T. DeMarco, "Specification Modelling," Software World, Vol. 12(4), pp. 2-9 (1981).

7. D. Bjorner, "The VDM Principles of Software Specification and Program Design," pp. 44-74, in Lecture Notes in Computer Science #107: Formalization of Programming Concepts, An International Colloquium, ed. I. Ramos, Springer-Verlag (1981).

8. P. E. Lauer, M. W. Shields, and J. Y. Cotronis, "Formal Behavioural Specifications of Concurrent Systems Without Globality Assumptions," pp. 115-151, in Lecture Notes in Computer Science #107: Formalization of Programming Concepts, An International Colloquium, ed. J. Diaz and I. Ramos, Springer-Verlag (1981).

9. S. W. Smoliar, "Operational Requirements Accomodation in Distributed System Design," IEEE Transactions on Software Engineering, Vol. 7(6), pp. 531-537 (November 1981).

10. A. J. Blikle, "On the Development of Correct Specified Programs," IEEE Transactions on Software Engineering, Vol. 7(5), pp. 519-527 (September 1981).

11. J. A. Goguen, "More Thoughts on Specification and Verification," SIGSOFT Software Engineering Notes, Vol. 6(3), pp. 38-41 (July 1981).

12. W. Swartout and R. Balzer, "On the Inevitable Intertwiningof Specification and Implementation," Communications of the ACM, Vol. 25(7), pp. 438-440 (July 1982).

13. P. E. Lauer, "Synchronization of Concurrent Processes without Globality Assumptions," SIGPLAN Notices, Vol. 16(9), pp. 66-80 (September 1981).

14. P. E. Lauer, P. R. Torrigiani, and M. W. Shields, "COSY: A System Specification Language Based on Paths and Processes," Acta Informatica, Vol. 12, pp. 109-158 (1979).

15. J. R. Abrial and S. A. Schuman, "Non-Deterministic System Specification," pp. 34-50, in _Lecture Notes in Computer Science #70: Semantics of Concurrent Computation,_ ed. G. Kahn, Springer-Verlag (1979).

16. N. A. Lynch and M. J. Fischer, "On Describing the Behavior of Distributed Systems," pp. 147-171, in _Lecture Notes in Computer Science #70: Semantics of Concurrent Computation,_ ed. G. Kahn, Springer-Verlag (1979).

17. J. S. Schwarz, "Denotational Semantics of Parallelism," pp. 191-202, in _Lecture Notes in Computer Science #70: Semantics of Concurrent Computation,_ ed. G. Kahn, Springer-Verlag (1979).

18. C. Hewitt, G. Attardi, and H. Lieberman, "Specifying and Proving Properties of Guardians for Distributed Systems," pp. 316-336, in _Lecture Notes in Computer Science #70: Semantics of Concurrent Computation,_ ed. G. Kahn, Springer-Verlag (1979).

19. P. Zave, "Testing Incomplete Specifications of Distributed Systems," _Proceedings of the ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing,_ pp. 42-48 (August 18-20, 1982).

20. P. Zave, "An Operational Approach to Requirements Specification of Embedded Systems," _IEEE Transactions on Software Engineering,_ Vol. SE-8(3), pp. 250-269 (May 1982).

BIBLIOGRAPHY ON DESIGN METHODOLOGIES

1. C. Alexander, in Notes on Synthesis of Form, Harvard University Press, Cambridge, Mass. (1964).

2. M. W. Alford and I. F. Burns, "R-Nets: A Graph Model for Real Time Software Requirements," in Proceedings 1976 MRI Conference Computer Software Engineering Reliability, Management, and Design (Apr. 1976).

3. S. Alter, "A Model for Automating File and Program Design in Business Applications Systems," Communications of the ACM, Vol. 22(6), p. 345 (June 1979).

4. M. Arisawa, "Evaluation of Design Methodologies: A Proposal," Software Engineering Notes, Vol. 7(1), p. 60 (Jan. 1982).

5. F. T. Baker, "Chief Programmer Team Management of Production Programming," IBM Systems Journal, Vol. 11(1), p. 56 (1972).

6. F. T. Baker, "Structured Programming in a Production Programming Environment," IEEE Transactions on Software Engineerin, Vol. 1, p. 241 (June 1975).

7. V. Basili and A. J. Turner, "Iterative Enhancement: A Practical Technique for Software Development," IEEE Transactions on Software Engineering, Vol. 1(4), p. 390 (Dec. 1975).

8. V. R. Basili and R. W. Reiter, "A Controlled Experiment Quantitatively Comparing Software Development Approaches," IEEE Transactions on Software Engineering, Vol. 7(3), p. 299 (May 1981).

9. L. A. Belady and M. M. Lehman, "Characteristics of Large Systems," in Proceedings Research Directions in Software Technology, Brown University (Oct. 1977).

10. G. D. Bergland and R. D. Gordon, pp. 1-14, in Tutorial: Software Design Strategies, IEEE Computer Society Press, Silver Spring, Md. (1979).

11. G. D. Bergland, "A Guided Tour of Program Design Methodologies," IEEE Computer, Vol. 14(10), p. 13 (Oct. 1981).

12. B. Boehm, R. McClean, and D. Urfrig, "Some Experiences with Automated Aids to the Design of Large Scale Reliable Software," pp. 105, in International Conference on Reliable Software, ACM (1975).

13. B. Boehm, "Seven Basic Principles of Software Engineering," in Infotech State of the Art Report on Software Engineering Techniques, Infotech International Ltd., Maidenhead UK (1976).

14. B. W. Boehm, "Software Engineering," IEEE Transactions on Computers, Vol. 25(12) (1976).

15. G. Booch, "Describing Software Design in Ada," Sigplan Notices, Vol. 16(9), p. 42 (Sept. 1981).

16. G. Booch, "Object Oriented Design," Ada Letters, Vol. I(3), p. 64 (Mar.-Apr. 1982).

17. D. L. Boyd and A. Pizzarello, "Introduction to the WELLMADE Design Methodology," _IEEE Transactions on Software Engineering,_ Vol. 4(4), p. 276 (Jul. 1978).

18. F. P. Brooks, in _The Mythical Man Month,_ Addison-Wesley (1975).

19. G. Bucci and D. N. Streeter, "A Methodology for the Design of Distributed Information Systems," _Communications of the ACM,_ Vol. 22(4), p. 233 (Apr. 1979).

20. J. N. Buxton and B. Randall, in _Software Engineering Techniques,_ NATO Scientific Affairs Division, Brussels (1970).

21. J. N. Buxton, "Software Engineering," pp. 23-28, in _Programming Methodology,_ ed. D. Gries, Springer-Verlag, N.Y. (1978).

22. S. H. Caine and E. K. Gordon, "PDL - A Tool for Software Design," pp. 271, in _Proceedings 1975 AFIPS NCC_ (Vol. 44).

23. D. R. Chand and S. B. Yadav, "Logical Construction of Software," _Communications of the ACM,_ Vol. 23(10), p. 546 (Oct. 1980).

24. D. Coleman, J. W. Hughes, and M. S. Powell, "A Method for the Syntax Directed Design of Multiprograms," _IEEE Transactions on Software Engineering,_ Vol. 7(2), p. 189 (Mar. 1981).

25. D. Comer, "Principles of Program Design Induced from Experience with Small Public Programs," _IEEE Transactions on Software Engineering,_ Vol. 7(2), p. 169 (MAr. 1981).

26. R. Conway, in _A Primer on Disciplined Programming,_ Winthrop (1978).

27. D. W. Crockett, "Triform Programs," Communications of the ACM, Vol. 24(6), p. 344 (June 1981).

28. O. J. Dahl, E. W. Dijkstra, and C. A. R. Hoare, in Structured Programming, Associated Press (1972).

29. C. G. Davis and C. R. Vick, "The Software Development System," IEEE Transactions on Software Engineering, Vol. 3(1), p. 69 (Jan. 1977).

30. R. A. Demers, "System Design for Usability," Communications of the ACM, Vol. 24(8), p. 494 (Aug. 1981).

31. E. W. Dijkstra and D. Gries, "Introduction to Programming Methodology," in Ninth Institute in Computer Science, University of California, Santa Cruz (Aug. 1979).

32. E. W. Dijkstra, in A Discipline of Programming, Prentice-Hall (1976).

33. E. W. Dijkstra, "The Humble Programmer," Communications of the ACM (Oct. 1972).

34. T. A. Dolotta and J. R. Mashey, "An Introduction to the Programmer's Workbench," pp. 164, in Proceedings of the Second International Conference on Software Engineering (1976).

35. L. E. Druffel, "The Potential Effect of Ada on Software Engineering in the 1980's," Software Engineering Notes, Vol. 7(3), p. 5 (July 1982).

36. A. G. Duncan and J. S. Hutchison, "Using Ada for Industrial Embedded Microprocessor Applications," Sigplan Notices, Vol. 15(11), p. 26 (Nov. 1980).

37. J. C. Enos and R. L. vanTilburg, "Tutorial: Software Design," IEEE Computer, Vol. 14(2), p. 61 (Feb. 1981).

38. P. Freeman and A. I. Wassermann, Tutorial on Software Design Techniques, IEEE Computer Society (1977).

39. T. Gilb, "Evolutionary Development," Software Engineering Notes, Vol. 6(2), p. 17 (Apr. 1981).

40. T. Gilb, pp. 214-217, in Software Metrics, Winthrop.

41. T. Gilb, "System Attribute Specification: A Cornerstone of Software Engineering," Software Engineering Notes, Vol. 6(3), p. 78 (July 1981).

42. T. Gilb, in Technoscopes ((MANUSCRIPT)).

43. M. Hamilton and S. Zeldin, "Higher Order Software -- A Methodology for Defining Software," IEEE Transactions on Software Engineering, Vol. 2(1), p. 9 (Mar. 1976).

44. H. Hart, "Ada for Design: An Approach for Transitioning Industry Software Developers," Ada Letters, Vol. II(1), p. 50 (Jul.-Aug. 1982).

45. H. C. Heacox, "RDL: A Language for Software Development," Sigplan Notices, Vol. 14(12), p. 71 (Dec. 1979).

46. M. A. Jackson, in *Principles of Program Design*, Academic Press, London (1975).

47. M. A. Jackson, "The Jackson Design Methodology," in *Infotech State of the Art Report: Structured Programming*.

48. E. P. Jensen, in *1975 IR&D Structural Design Methodology -- Volume II*, Hughes Aircraft Co. (Dec. 1975).

49. R. W. Jensen and C. C. Tonies, in *Software Engineering*, Prentice Hall (1979).

50. R. W. Jensen, "Tutorial: Structured Programming," *IEEE Computer*, Vol. 14(3), p. 31 (Mar. 1981).

51. J. W. Johnson, "Software Design Techniques," in *Proceedings National Electronics Conference* (Oct. 1977).

52. B. Kumar and E. S. Davidson, "Computer Systems Design Using a Hierarchical Approach to Performance Evaluation," *Communications of the ACM*, Vol. 23(9), p. 511 (Sept. 1980).

53. R. F. Ling, "General Considerations on the Design of an Interactive System for Data Analysis," *Communications of the ACM*, Vol. 23(3), p. 147 (Mar. 1980).

54. R. C. Linger, H. D. Mills, and B. I. Witt, in *Structured Programming: Theory and Practice*, Addison-Wesley (1979).

55. B. Lint and T. Agerwala, "Communication Issues in the Design and Analysis of Parallel Algorithms," *IEEE Transactions on Software Engineering*, Vol. 7(2), p. 174 (Mar. 1981).

56. B. H. Liskov, "A Design Methodology for Reliable Software Systems," pp. 191-199, in AFIPS Conference Proceedings, Vol. 41 (FJCC 72) (1972).

57. M. MacLaren, "Evolving Toward Ada in Real Time Systems," Sigplan Notices, Vol. 15(11), p. 146 (Nov. 1980).

58. A. deMarcoe, in Structured Analysis and System Specification, Yourdon, Inc., N.Y. (1978).

59. B. Meyer, "Principles of Package Design," Communications of the ACM, Vol. 25(7), p. 419 (July 1982).

60. A. Mili, "Designing Software Without Backtracking, The Part of Dream and the Part of Reality," Software Engineering Notes, Vol. 7(1), p. 77 (Jan. 1982).

61. H. Mills and Dyer, in IBM Systems Journal (Apr. 1980).

62. G. J. Myers, in Reliable Software Through Composite Design, Petrocelli/Charter (1975).

63. P. Naur and B. Randall, in Software Engineering, NATO Scientific Affairs Division, Brussels (1969).

64. P. G. Neumann, "Experience with a Formal Methodology for Software Development," in Proceedings International Seminar on Software Engineering Applications (1980).

65. D. L. Parnas and Darringer, "SODAS and Methodology for System Design," pp. 449, in Proceedings AFIPS Fall Joint Computer Conference (1967).

66. D. L. Parnas, "Designing Software for Ease of Extension and Contraction," IEEE Transactions on Software Engineering, Vol. 5, p. 128 (MAr. 1979).

67. D. L. Parnas, "Information Distribution Aspects of Design Methodology," in Information Processing 1971, North Holland (1972).

68. D. L. Parnas, : Hierarchical Structure"" "On a "Buzzword": Hierarchical Structure," pp. 336, in IFIP 1974 (1974).

69. D. L. Parnas, "On the Criteria to be Used in Decomposing Systems into Modules," Communications of the ACM (Dec. 1972).

70. D. L. Parnas, "Software Engineering or Methods for the Multi-Person Construction of Multi-Version Programs," Lecture Notes in Computer Science (Programming Methodology), Vol. 23, pp. 225-235 (1975).

71. D. L. Parnas, "The Influence of Software Structure on Reliability," Sigplan Notices, Vol. 10(6) (June 1975).

72. A. V. Pohm and T. A. Smay, "Tutorial: Top-Down Design," IEEE Computer, Vol. 14(6), p. 65 (June 1981).

73. C. V. Ramamoorthy, Y. K. R. Mok, F. B. Bastani, G. H. Chin, and K. Suzuki, "Application of a Methodology for the Development and Validation of Reliable Process Control Software," IEEE Transactions on Software Engineering, Vol. 7(6), p. 537 (Nov. 1981).

74. J. Ramanathan and C. J. Shubra, "Modeling of Problem Domains for Driving Program Development Systems," pp. 28, in Proceedings of the Eighth Annual Symposium on Principles of Programming

Languages (1981).

75. T. L. Ramey, "Introduction to ERA Information Modeling," FR 79-70-873R, Hughes Aircraft Co. (June 1979).

76. L. Robinson, "The HDM Handbook," 4828, Computer Science Group SRI International (1979).

77. D. T. Ross, J. B. Goodenough, and C. A. Irvine, "Software Engineering: Processes, Principles and Goals," IEEE Computer, p. 62 (May 1975).

78. D. T. Ross and K. E. Schoman, "Structured Analysis for Requirements Definition," IEEE Transactions on Software Engineering, Vol. 3(1) (Jan. 1977).

79. P. E. Schilling, "A Program Model for Design," Software Engineering Notes, Vol. 6(2), p. 15 (Apr. 1981).

80. C. Shaw, "Structure Charts for Jackson Structured Programming," Software Engineering Notes, Vol. 7(1), p. 78 (Jan. 1982).

81. J. M. Spitzen, K. N. Levitt, and L. Robinson, "An Example of Hierarchical Design and Proof," Communications of the ACM, Vol. 21(12), p. 1064 (Dec. 1978).

82. W. P. Stephens, G. J. Myers, and L. L. Constantine, "Structured Design," IBM Systems Journal, Vol. 13(2), p. 115.

83. W. P. Stevens, G. J. Myers, and L. L. Constantine, "Structured Design," IBM Systems Journal, Vol. 13(2), p. 115 (1974).

84. S. A. Sutton and V. R. Basili, "The FLEX Software Design System: Designers Need Languages, Too," IEEE Computer, Vol. 14(11), p. 95 (Nov. 1981).

85. D. Teichroew and E. A. Hershey, "PSL/PSA: A Computer-aided Technique for Structured Documentation and Analysis of Information Processing Systems," IEEE Transactions on Software Engineering, Vol. 3(1), p. 41 (1977).

86. J. D. Warnier, pp. 1976, in Logical Construction of Programs, Van Nostrand Reinhold, N.Y..

87. A. I. Wassermann and S. Gutz, "The Future of Programming," Communications of the ACM, Vol. 25(3), p. 196 (Mar. 1982).

88. A. I. Wassermann, "Information System Design Methodology," Journal American Society Information Sciences, Vol. 31(1), p. 5 (Jan. 1980).

89. A. I. Wassermann, "Software Engineering: The Turning Point," IEEE Computer, Vol. 11(9), p. 30 (Sept. 1978).

90. A. I. Wassermann, "USE: A Methodology for the Design and Development of Interactive Information Systems," pp. 31, in Formal Models and Practical Tools for Information System Design, ed. H. J. Schneider, North Holland (1979).

91. D. W. Waugh, "Ada as a Design Language," IBM Software Engineering Exchange, Vol. 3(1) (Oct. 1980).

92. T. J. Wheeler, in Embedded Systems Design with Ada as the Systems Design Language, CORADCOM (1980).

93. R. R. Willis and E. P. Jensen, "Computer Aided Design of Software Systems," in Proceedings 4th International Conference on Software Engineering (Sept. 1979).

94. R. R. Willis, "DAS -- An Automated System to Support Design Analysis," in Proceedings 3rd International Conference on Software Engineering (May 1978).

95. J. W. Winchester, "Requirements Design Language," in Ph.D. Thesis, UCLA (June 1980).

96. N. Wirth, in Systematic Programming, Prentice Hall (1973).

97. N. Wirth, "On the Composition of Well-Structured Programs," Computing Surveys, Vol. 6(4), p. 247 (Dec. 1974).

98. N. Wirth, "Program Development by Stepwise Refinement," Communications of the ACM, Vol. 14(4), p. 221 (Apr. 1971).

99. S. S. Yau, C. C. Yang, and S. M. Shatz, "An Approach to Distributed Computing System Software Design," IEEE Transactions on Software Engineering, Vol. 7(4), p. 427 (July 1981).

100. E. Yourdon and L. Constantine, in Structured Design, Prentice-Hall (1979).

101. E. Yourdon, in Managing the Structured Techniques, Prentice-Hall (1979).

102. M. V. Zelkowitz, "Perspectives on Software Engineering," Computing Surveys, Vol. 10(2), p. 197 (June 1978).

103. "An Introduction to SADT," 9022-78, Softech, Inc., Waltham Mass. (Feb. 1976).

104. "HIPO: Design Aid and Documentation Tool," 20-9413-0, IBM Systems Research (1973).

105. in _Implications of Using Modular Programming,_ Central Computer Agency, Her Majesty's Stationery Office, London, UK (1973).

BIBLIOGRAPHY ON PARALLEL PROGRAMMING LANGUAGES

1. G. R. Andrews, "Synchronizing Resources," ACM Transactions on Programming Languages and Systems, Vol. 3(4), pp. 405-431 (Oct. 1981).

2. G. R. Andrews, "The Design of a Message Switching System : An Application and Evaluation of Modula," IEEE Transactions on Software Engineering, Vol. 5(2) (March 1979).

3. Arvind,, K. P. Gostelow, and W. Plouffe, Indeterminacy, Monitors and Dataflow, University of California at Irvine (March 1977).

4. R. M. Balzer, "An Overview of the ISPL Computer System Design," Communications of the ACM, Vol. 16(2), pp. 117-122 (Feb. 1973).

5. A. Bernstein, "Output Guards and Nondeterminism in 'Communicating Sequential Processes'," ACM Transactions on Programming Languages and Systems, Vol. 2(2), pp. 234-238 (Apr. 1980).

6. J. vandeBose, R. Plasmeijer, and J. Stroet, "Process Communication Based on Input Specifications," ACM Transactions on Programming Languages and Systems, Vol. 3(3), pp. 224-250 (July 1981).

7. P. Brinch Hansen, in Edison Programs, Computer Science Department, University of Southern California (Sept. 1980).

8. P. Brinch Hansen, in Edison, A Multiprocessor Language, Computer Science Department, University of Southern California (Sept.

1980).

9. P. Brinch Hansen, in _The Design of Edison_, Computer Science Department, University of Southern California (Sept. 1980).

10. P. Brinch Hansen, "Distributed Processes: A Concurrent Programming Concept," _Communications of the ACM_, Vol. 21(11), pp. 934-940 (Nov. 1978).

11. P. Brinch Hansen, "Experience with Modular Concurrent Programming," _IEEE Transactions on Software Engineering_, Vol. 3(2), pp. 156-159 (March 1977).

12. P. Brinch Hansen, "NETWORK: A Multiprocessor Program," _IEEE Transactions on Software Engineering_, Vol. 4(3), pp. 194-199 (May 1978).

13. P. Brinch Hansen, "The Programming Language Concurrent Pascal," _IEEE Transactions on Software Engineering_, Vol. 1(21), pp. 199-207 (June 1975).

14. R. E. Bryant and J. B. Dennis, "Concurrent Programming," pp. 584-610, in _Research Directions in Software Technology_, ed. P. Wegner (1980).

15. R. H. Campbell and A. N. Habermann, "The Specification of Process Synchronization by Path Expressions," _Lecture Notes in Computer Science_ (1974).

16. R. H. Campbell and Kolstad, "An Overview of PATH PASCAL's Design and PATH PASCAL User Manual," _SIGPLAN Notices_, Vol. 15(9), pp. 13-24 (Sept. 1980).

17. K. M. Chandy and J. Misra, "A Simple Model of Distributed Programs Based on Implementation-Hiding and Process Autonomy," SIGPLAN Notices, Vol. 15(7), pp. 26-35 (Jul-Aug. 1980).

18. K. M. Chandy and J. Misra, An Axiomatic Proof Technique for Networks of Communicating Processes, Department of Computer Science, University of Texas, Austin (1978).

19. K. M. Chandy and J. Misra, "Deadlock Absence Proofs for Networks of Communicating Processes," Info Processing Letters, Vol. 9(4), pp. 185-189 (Nov. 1979).

20. E. M. Clarke, "Synthesis of Resource Invariants for Concurrent Programs," ACM Transactions on Programming Languages and Systems, Vol. 2(3) (July 1980).

21. J. Dennis, D. P. Misunas, and C. K. Leung, A Highly Parallel Processor Using a Data Flow Machine Language. Jan. 1977.

22. E. W. Dijkstra, "Co-operating Sequential Processes," in Programming Languages, ed. F. Guneys, Academic Press (1968).

23. S. Even, "Parallelism in Tape Sorting," Communications of the ACM, Vol. 17(4) (April 1974).

24. J. Feldman, "High Level Programming for Distributed Computing," Communications of the ACM, Vol. 22(6), pp. 353-368 (June 1979).

25. D. P. Friedman and D. S. Wise, "A Note on Conditional Expressions," Communications of the ACM, Vol. 21(11), pp. 931-933 (Nov. 1978).

26. D. Good, R. Cohen, and J. Keeton-Williams, "Principles of Proving Concurrent Programs in GYPSY," in Certifiable Minicomputer Project, University of Texas (Jan. 1979).

27. D. Good, R. Cohen, and L. W. Hunter, "A Report on the Development of GYPSY," in Certifiable Minicomputer Project, University of Texas (Oct. 1978).

28. D. Good and R. Cohen, "Verifyable Communications Processing in GYPSY," in Certifiable Minicomputer Project, University of Texas (June 1978).

29. D. Gries, "An Excercise in Proving Parallel Programs Correct," Communications of the ACM, Vol. 20(12) (Dec. 1977).
Corrigendum: CACM vol:21, #12, Dec. 1978

30. A. N. Habermann, "Synchronization of Communicating Processes," Communications of the ACM, Vol. 21(8), pp. 666-677 (Aug. 1978).

31. D. S. Hirschberg, "Fast Parallel Sorting Algorithms," Communications of the ACM, Vol. 21(8) (Aug. 1978).

32. C. A. R. Hoare, "Communicating Sequential Processes," Communications of the ACM, Vol. 21(8), pp. 666-677 (Aug. 1978).

33. C. A. R. Hoare, "Monitors: An Operating System Structuring Concept," Communications of the ACM, Vol. 17(10), pp. 549-557 (Oct. 1974).

34. C. A. R. Hoare, "Towards a Theory of Parallel Programming," International Seminar on Operating Systems Techniques, Belfast, Northern Ireland (Aug. 1971).

35. J. H. Howard, "Proving Monitors," Communications of the ACM, Vol. 19(10) (Oct. 1976).

36. J. Ichbiah, "Chapter 11 and Chapter 9," in Ada Rationale and Ada Reference Manual (1980, 1981).

37. E. L. Jacks, "An Iterative Cell Processor for the ASP Language," pp. 147-172, in Associative Information Techniques, American Elsevier, New York (1971).

38. R. M. Keller, "Formal Verification of Parallel Programs," Communications of the ACM, Vol. 19(7) (July 1976).

39. R. B. Kieburtz and A. Silberschatz, "Comments on 'Communicating Sequential Processes'," ACM Transactions on Programming Languages and Systems, Vol. 1(2), pp. 218-225 (Oct. 1979).

40. L. Lamport, "The Parallel Execution of DO-Loops," Communications of the ACM, Vol. 17(2) (Feb. 1974).

41. B. Lampson and D. Redell, "Experience with Processes and Monitors in MESA," Communications of the ACM, Vol. 23(2), pp. 105-117 (Feb. 1980).

42. P. E. Lauer, "Synchronization of Concurrent Processes without Globality Assumptions," SIGPLAN Notices, Vol. 16(9), pp. 66-80 (Sept. 1981).

43. D. Lawrie, "GLYPNIR A Programming Language for the ILLIAC IV," Communications of the ACM (March 1975).

44. R. J. Lipton, "Reduction : A Method of Proving Properties of Parallel Programs," Communications of the ACM, Vol. 18(12) (Dec. 1975).

45. B. Liskov, Primitives for Distributed Computing, MIT Computation Structures Group (May 1979).

46. R. E. Millstein, "Control Structures in ILLIAC IV Fortran," Communications of the ACM, Vol. 16(10) (Oct. 1973).

47. S. Owicki and D. Gries, "Axiomatic Proof Techniques for Parallel Programs," Acta Informatica (June 1976).

48. H. A. Schutz, "On the Design of a Language for the Programming Real-Time Concurrent Processes," IEEE Transactions on Software Engineering, Vol. 5(3) (May 1979).

49. A. Silberschatz, "On The Synchronization Mechanism of the Ada Language," SIGPLAN Notices, Vol. 16(2), pp. 96-103 (Feb. 1981).

50. A. Silberschatz, Port-Directed Communication, University of Texas at Dallas (Mar. 1979).

51. A. Silbershatz, R. B. Kieburtz, and A. J. Bernstein, "Extending Concurrent Pascal to Allow Dynamic Resource Management," IEEE Transactions on Software Engineering, Vol. 3(3) (May 1977).

52. J. Stroet, "An Alternative to the Communication Primitives in Ada," SIGPLAN Notices, Vol. 15(12), pp. 62-74 (Dec. 1980).

53. N. Wirth, "Design and Implementation of Modula," Software-Practice and Experience, Vol. 7, pp. 67-84 (1977).

54. N. Wirth, "The Use of Modula," *Software-Practice and Experience*, Vol. 7, pp. 37-65 (1977).

BIBLIOGRAPHY ON TESTING METHODOLOGIES

1. S. B. Akers, "Test Generation Techniques," Computer, Vol. 13 (1980).

2. F. Bazzichi and I. Spadafora, "An Automatic Generator for Compiler Testing," IEEE Transactions on Software Engineering, Vol. 8(4), p. 343 (Jul. 1982).

3. W. E. Boebert, "The Analytic Verification of Flight Software: A Case Study," Proceedings of IEEE 1978 National Aerospace and Electr. Conference (1978).

4. R. S. Boyer, B. Elspas, and K. N. Levitt, "SELECT - A Formal System for Testing and Debugging Programs by Symbolic Execution," Proceedings 1975 International Conference Reliable Software, p. 234 (Apr. 1975).

5. M. A. Branstad, "Validation, Verification, and Testing for the Individual Programmer," Computer, Vol. 13(12), p. 24 (Dec 1980).

6. M. Brooks, "Determining Correctness by Testing," STAN-CS-80-804, Stanford University, Stanford, CA (1980).

7. J. R. Brown and M. Lipow, "Testing for Software Reliability," pp. 518-527, in Proceedings 1975 International Conference on Reliable Software (Apr. 1975).

8. T. A. Budd, R. A. DeMillo, R. J. Lipton, and F. G. Sayward, "Theoretical and Empirical Studies on Using Program Mutation to Test the Functional Correctness of Programs," in Proceedings ACM Symposium on Principles of Programming Languages (1980).

9. T. A. Budd, R. J. Lipton, R. A. DeMillo, and F. G. Sayward, "Mutation Analysis," 155, Department of Computer Science, Yale University, New Haven, CT (Apr. 1979).

10. T. A. Budd, R. A. DeMillo, R. J. Lipton, and F. G. Sayward, "The Design of a Prototype Mutation System for Program Testing," Proceedings 1978 National Computer Conference.

11. T. S. Chow, "Testing Software Design Modeled by Finite State Machines," IEEE Transactions on Software Engineering, Vol. SE-4, p. 178 (May 1978).

12. L. A. Clarke, J. Hassell, and D. J. Richardson, "A Close Look at Domain Testing," IEEE Transactions on Software Engineering, Vol. 8(4), p. 380 (Jul. 1982).

13. L. A. Clarke, "A System to Generate Test Data and Symbolically Execute Programs," IEEE Transactions Software Engineering, Vol. 2(3), p. 227 (1976).

14. L. A. Clarke, "Testing: Achievements and Frustrations," Proceedings COMSAC 78, p. 310, IEEE (Nov 1978).

15. E. I. Cohen, "A Finite Domain-Testing Strategy for Comuter Program Testing," Ph.D. Dissertation, Ohio State University (Jun 1978).

16. J. A. Darringer, "Application of Symbolic Execution to Program Testing," Computer, Vol. 11(4), p. 51 (Apr 1978).

17. R. A. DeMillo and R. J. Lipton, "A Probabilistic Remark on Algebraic Program Testing," Information Processing Letters, Vol. 7 (June 1978).

18. R. A. DeMillo, R. J. Lipton, and F. G. Sayward, "Hints on Test Data Selection: Help fot the Practicing Programmer," IEEE Computer, Vol. 11(4), p. 34 (April 1978).

19. A. G. Duncan and J. S. Hutchison, "Using Attributed Grammars to Test Designs and Implementations," in Proceedings 5th IEEE International Conference on Software Engineering (1981).

20. A. Endres, "An Analysis of Errors and their Causes in System Programs," pp. 327-336, in Proceedings International Conference on Reliable Software (1975).

21. R. E. Fairley, "Ada Debugging and Testing Support Environments," SIGPLAN Notices, Vol. 15(11), p. 16 (Nov 1980).

22. R. E. Fairley, "An Experimental Program Testing Facility," IEEE Transactions on Software Engineering, Vol. SE-1(4), p. 350 (1975).

23. R. E. Fairley, "Dynamic Testing of Simulation Software," Proceedings Summer Computer Simulation Conference (Jul 1976).

24. R. E. Fairley, "Tutorial: Static Analysis and Dynamic Testing of Computer Software," Computer, Vol. 11(4), p. 14 (Apr 1978).

25. E. H. Forman and N. D. Singpurwalla, "An Empirical Stopping Rule for Debugging and Testing Computer Software," Journal of the American Statistical Association, Vol. 72, pp. 750-757 (Dec. 1977).

26. E. H. Forman and N. D. Singpurwalla, "Optimal Time Intervals for Testing Hypotheses on Computer Software Errors," IEEE Transactions on Reliability, Vol. R-28, pp. 250-253 (Aug. 1979).

27. K. A. Foster, "Error Sensitive Test Cases Analysis (ESTCA)," IEEE Transactions on Software Engineering, Vol. SE-6(3), p. 258 (May 1980).

28. H. N. Gabow, S. N. Maheshwari, and L. J. Osterweil, "On Two Problems in the Generation of Program Test Paths," IEEE Transactions on Software Engineering, Vol. SE-2, p. 227 (Sept 1976).

29. C. Gannon, "Error Detection Using Path Testing and Static Analysis," Computer, Vol. 12(8), p. 26 (Aug 1979).

30. C. Gannon, "JAVS: A Jovial Automated Verification System," Proceedings COMSAC 78, p. 539, IEEE (Nov 1978).

31. M. Geller, "Test Data as an Aid in Proving Program Correctness," Communications of the ACM, Vol. 21(5), p. 368 (May 1978).

32. J. B. Goodenough and S. L. Gerhart, "Correction to 'Toward a Theory of Test Data Selection'," IEEE Trns. on Software Engineering, Vol. SE-1 (Dec 1975).

33. J. Goodenough and S. L. Gerhart, "Toward a Theory of Test Data Selection," IEEE Transactions on Software Engineering, Vol. 3 (1977).

34. J. B. Goodenough and S. L. Gerhart, "Toward a Theory of Testing: Data Selection Criteria," pp. 44, in _Current Trends in Programming Methodology_, ed. R. T. Yeh, Prentice Hall, Englewood Cliffs NJ (1977).

35. J. B. Goodenough, "A Survey of Program Testing Issues," pp. 316, in _Research Directions in Software Technology_, ed. P. Wegner, MIT Press (1979).

36. J. B. Goodenough, "Toward a Theory of Test Data Selection," _IEEE Transactions Software Engineering_, Vol. SE-1(2), p. 156 (1975).

37. T. G. Hallin, "Toward a Better Method of Testing Software," _Proceedings COMSAC 78 Computer Software and Applications Conference_, p. 153, IEEE (Nov 1978).

38. R. G. Hamlet, "Testing Programs with the Aid of a Compiler," _IEEE Transactions Software Engineering_, Vol. SE-3(4), p. 279 (Jul 1977).

39. K. V. Hanford, "Automatic Generation of Test Cases," _IBM Systems Journal_, Vol. 9, pp. 242-257 (Dec. 1970).

40. R. D. Hartwick, "Test Planning," _AFIPS Conference Proceedings 1977 National Computer Conference_, Vol. 46, p. 285.

41. M. A. Hennell, "Approaches to Testing," 1980, Department of Computing and Statistical Science, University of Liverpool, Liverpool, England (1980).

42. W. C. Hetzel, in _Program Test Methods_, Prentice Hall, Englewood Cliffs, NJ (1973).

43. E. H. Horman, "Optimal Time Intervals for Testing Hypotheses on Computer Software Errors," IEEE Transactions Reliability, Vol. R-28(3), p. 250 (Aug 1979).

44. W. E. Howden and P. Eichhorst, "Proving Program Properties From Traces," in Software Testing and Verification Techniques, ed. E. Miller & W. E. Howden, IEEE (1978).

45. W. E. Howden, "An Evaluation of the Effectiveness of Symbolic Testing," Software-Practice and Experience, Vol. 8, p. 381 (1978).

46. W. E. Howden, "Algebraic Program Testing," Acta Informatica, Vol. 10 (1978).

47. W. E. Howden, "Applicability of Software Validation Techniques to Scientific Programs," ACM Transactions on Programming Languages and Systems, Vol. 2 (1980).

48. W. E. Howden, "Fractional Testing and Design Abstractions," Journal of Systems and Software, Vol. 1(4), p. 307 (1980).

49. W. E. Howden, "Functional Program Testing," Proceedings COMSAC 78, p. 315, IEEE (Nov 1978).

50. W. E. Howden, "Functional Program Testing," IEEE Transactions on Software Engineering, Vol. 6 (1980).

51. W. E. Howden, "Methodology for the Generation of Program Test Data," IEEE Transactions on Computers, Vol. C-24(5), p. 554 (May 1975).

52. W. E. Howden, "Reliability of Symbolic Evaluation," Proceedings Computer Software and Applications Conference, p. 442, IEEE Computer Society (Nov 1977).

53. W. E. Howden, "Reliability of the Path Analysis Testing Strategy," IEEE Transactions Software Engineering, Vol. SE-2(3), p. 208 (Sept 1976).

54. W. E. Howden, "Symbolic Testing and the DISSECT Symbolic Evaluation System," IEEE Transactions on Software Engineering, Vol. SE-3, p. 266 (Jul 1977).

55. W. E. Howden, "Theoretical and Empirical Studies of Program Testing," IEEE Transactions on Software Engineering, Vol. 4(4), p. 293 (Jul. 1978).

56. W. E. Howden, "Weak Mutation Testing and Completeness of Test Sets," IEEE Transactions on Software Engineering, Vol. 8(4), p. 371 (Jul. 1982).

57. J. C. Huang, "An APproach to Program Testing," Computing Surveys, Vol. 7(3), pp. 113-128 (Sept 1975).

58. J. C. Huang, "Instrumenting Programs for Symbolic Trace Generation," Computer, Vol. 13(12), p. 17 (Dec 1980).

59. J. C. Huang, "Program Instrumentation and Software Testing," IEEE Computer, Vol. 11(4), p. 25 (April 1978).

60. J. C. King, "A New Approach to Program Testing," Proceedings 1975 International Conference on Reliable Software, Vol. IEEE Cat. No. 75CH0940-7CSR, p. 228 (Apr 1975).

61. J. C. King, "Symbolic Execution and Program Testing," Communications of the ACM, Vol. 19(7), p. 385 (Jul 1976).

62. K. W. Krause, "Optimal Software Test Planning Through Automated Network Analysis," IEEE Symposium on Computer Software Reliability, p. 18 (Apr 1973).

63. J. W. Laski, "A Heirarchical Approach to Program Testing," SIGPLAN Notices, Vol. 15(1), p. 77 (June 1980).

64. R. J. Lipton and F. G. Sayward, "The Status of Research on Program Mutation Testing," in Digest of Papers, Workshop on Software Testing and Test Documentation (Dec. 1978).

65. E. Miller, M. P. Paige, J. P. Benson, and W. R. Wisehart, "Structural Techniques of Program Validation," in Proceedings IEEE COMPCON (1974).

66. E. F. Miller, in Program Testing Techniques, IEEE Computer Society, Long Beach, Calif. (1977).

67. E. F. Miller, "Program Testing," IEEE Computer, Vol. 11(4), p. 10 (April 1978).

68. E. F. Miller, "Program Testing Tools - A Survey," MIDCON Proceedings (Nov 1977).

69. E. F. Miller, "Program Testing: Art Meets Theory," Computer, Vol. 10(7), p. 42 (Jul 1977).

70. P. B. Moranda, "Limits to Program Testing with Random Number Inputs," pp. 521-526, in Proceedings COMPSAC 1978 (Nov. 1978).

71. G. J. Myers, in *The Art of Software Testing*, Wiley, Chichester, England (1979).

72. G. J. Myers, "A Controlled Experiment in Program Testing and Walkthroughs/Inspection," *Communications of the ACM*, Vol. 21(9), p. 760 (Sept 1978).

73. E. Nelson, "Estimating Software Reliability from Test Data," *Microelectronics and Reliability*, Vol. 17, pp. 67-74 (1978).

74. L. J. Osterweil, in *Allegations as Aids to Static Program Testing*, Department of Computer Science Report, Univ. Colorado, Boulder, Co..

75. M. Paige, "An Analytical Approach to Software Testing," *Proceedings COMSAC 78*, p. 527, IEEE (Nov 1978).

76. M. R. Paige and J. P. Benson, "The Use of Software Probes in Testing FORTRAN Programs," *Computer*, Vol. 7(4), p. 40 (Jul 1974).

77. D. J. Panzel, "Automated Revision of Formal Test Procedures," *3rd International Conference on Software Engineering*, p. 320, IEEE (1978).

78. D. J. Panzel, "Automatic Software Test Drivers," *Computer*, Vol. 11(4), p. 44 (Apr 1978).

79. D. J. Panzel, "Test Procedures: A New Approach to Software Verification," *Proceedings Second International Conference on Software Engineering*, p. 477 (Oct 1976).

80. S. Pimont and J. C. Rault, "A Software Reliability Assessment Based on a Structural and Behavioral Analysis of Programs," in _Proceedings 2nd IEEE International Conference on Software Engineering_ (1976).

81. J. A. Rader, "VEVA- A Coherent Static Code Analysis System," _Conference Record of the 12th Asimolar Conference on Circuits, Systems, and Computers_, IEEE (Nov 1978).

82. C. V. Ramamoorthy and F. B. Bastani, "Software Reliability -- Status and Perspectives," _IEEE Transactions on Software Engineering_, Vol. 8(4), p. 354 (Jul. 1982).

83. C. V. Ramamoorthy, K. H. Kim, and W. T. Chen, "Optimal Placement of Software Monitors Aiding Systematic Testing," _IEEE Transactions on Software Engineering_, Vol. SE-1(4), p. 403 (Dec 1975).

84. C. V. Ramamoorthy, S. F. Ho, and W. T. Chen, "On the Automated Generation of Program Test Data," _IEEE Transactions on Software Engineering_, Vol. SE-2(4), p. 293 (Dec 1976).

85. C. V. Ramamoorthy and S. F. Ho, "Testing Large Software with Automated Software Evaluation Systems," _IEEE Transactions Software Engineering_, Vol. 1(1), p. 46 (March 1975).

86. R. J. Reifer, "Software Failure Modes and Effect Analysis," _IEEE Transactions on Reliability_, Vol. R-28, pp. 247-249 (Aug. 1979).

87. D. J. Richardson and L. A. Clarke, "A Partition Analysis Method to Increase Program Reliability," pp. 244-253, in _Proceedings 5th International Conference on Software Engineering_ (Mar. 1981).

88. G. J. Schick and R. W. Wolverton, "Analysis of Error Processes in Computer Software," pp. 337-346, in Proceedings International Conference on Reliable Software (1975).

89. N. R. Schneidewind, "Applications of Program Graphs and Complexity Analysis to Software Development and Testing," IEEE Transactions Reliability, Vol. R-28(3), p. 192 (Aug 1979).

90. M. L. Shooman and M. I. Bolsky, "Types, Distribution, and Test and Correction Times for Programming Errors," pp. 347-357, in Proceedings International Conference on Software Reliability (1975).

91. M. L. Shooman, "Operational Testing and Software Reliability Estimation During Program Development," pp. 51-57, in Record 1973 IEEE Symposium on Computer Software Reliability (May 1973).

92. A. R. Sorkowitz, "Certification Testing: A Procedure to Improve the Quality of Software Testing," Computer, Vol. 12(8), p. 20 (Aug 1979).

93. L. G. Stucki, "A Prototype Automatic Program Testing Tool," AFIPS Conference Proceedings, Vol. 41, p. 829 (1972).

94. R. N. Taylor, Integrated Testing and Verification System for Research Flight Software -Design Document, Boeing Computer Services (Feb 1979).

95. A. E. Tucker, The Correlation of Computer Program Quality with Testing Effort. Jan 1965.

96. A. M. Turing, "Checking a Large Routine," in Report of a Conference on High Speed Automatic Calculating Machines, University

Computing Lab., Cambridge (Jan. 1950).

97. U. Voges, L. Gmeiner, Amschler, and A. vonMayrhauser, "SADAT - An Automated Testing Tool," IEEE Transactions on Software Engineering, Vol. SE-6(3), p. 286.

98. E. J. Weyuker and T. J. Ostrand, "Theories of Program Testing and the Application of Revealing Subdomains," pp. 1-18, in Digest of a Workshop on Software Testing and Test Documentation (Dec. 1978).

99. E. J. Weyuker and T. J. Ostrand, "Theories of Program Testing and the Application of Revealing Subdomains," IEEE Transactions on Software Engineering, Vol. SE-6(3), p. 236 (May 1980).

100. L. J. White and E. I. Cohen, "A Domain Strategy for Computer Program Testing," IEEE Transactions on Software Engineering, Vol. SE-6(3), p. 247 (May 1980).

101. L. J. White, F. C. Teng, H. C. Kuo, and D. W. Coleman, An Error Analysis of the Domain Testing Strategy, Computer and Information Science Research Center, Ohio State Univ., Columbus (Aug 1978).

102. M. R. Woodward, D. Hedley, and M. A. Hennell, "Experience with Path Analysis and Testing of Programs," IEEE Transactions on Software Engineering, Vol. SE-6(3), p. 278 (May 1980).

103. S. J. Zeil and L. J. White, "Sufficient Test Sets for Path Analysis Testing Strategies," in Proceedings 5th IEEE International Conference on Software Engineering (1981).

104. "A Method of Test Structuring for Debugging Control Programs," Program and Computing Software, Vol. 6(2), p. 98.

BIBLIOGRAPHY ON STATIC ANALYSIS

1. P. W. Abrahams and L. A. Clarke, "Compile-Time Analysis of Data List-Format Correspondences," IEEE Transactions on Software Engineering, Vol. 5, pp. 612-617 (Nov. 1979).

2. F. E. Allen and J. Cocke, "A Program Data Flow Analysis Procedure," Communications of the ACM, Vol. 19(3), pp. 137-147 (Mar. 1976).

3. J. M. Barth, "A Practical Interprocedural Data Flow Analysis Algorithm," Communications of the ACM, Vol. 21(9), pp. 724-736 (Sept. 1978).

4. J. C. Browne and D. B. Johnson, "FAST: A Second Generation Program Analysis System," in Tutorial: Auttomated Tools for Software Engineering, ed. E. Miller, IEEE (EHO 150-3 COMPSAC 79) (1979).

5. L. D. Fosdick and L. J. Osterweil, "Data Flow Analysis in Software Reliability," ACM Computing Surveys, Vol. 8(3), pp. 305-330 (Sept. 1976).

6. W. H. Harrison, "Compiler Analysis of the Value Ranges for Variables," IEEE Transactions on Software Engineering, Vol. 3, pp. 243-250 (MAy 1977).

7. M. S. Hecht and J. D. Ullman, "A Simple Algorithm for Global Data Flow Analysis Problems," SIAM Journal of Computing, Vol. 4(4), pp.

519-532 (Dec. 1975).

8. J. C. Huang, "Detection of Data Flow Anomaly Through Program Instrumentation," IEEE Transactions on Software Engineering, Vol. 5(3), pp. 226-236 (May 1979).

9. L. J. Osterweil and L. D. Fosdick, "DAVE -- A Validation Error Detection and Documentation System for Fortran Programs," Software-Practice and Experience, Vol. 6, pp. 473-486 (1976).

10. C. V. Ramamoorthy and S. F. Ho, "Testing Large Software with Automated Software Evaluation Systems," IEEE Transactions on Software Engineering, Vol. 1, pp. 46-58 (Mar. 1975).

11. R. N. Taylor and L. J. Osterweil, "Anomaly Detection in Concurrent Software by Static Data Flow Analysis," IEEE Transactions on Software Engineering, Vol. 6, pp. 265-278 (May 1980).

12. U. Voges, L. Gmeiner, and A. A. vonMayrhauser, "SADT -- An Automated Testing Tool," IEEE Transactions on Software Engineering, Vol. 6, pp. 286-290 (May 1980).

BIBLIOGRAPHY ON SYMBOLIC EXECUTION

1. L. A. Clarke, "A System to Generate Test Data and Symbolically Execute Programs," IEEE Transactions on Software Engineering, Vol. 2(3), pp. 215-222 (Sept. 1976).

2. S. L. Hantler and J. C. King, "An Introduction to Proving the Correctness of Programs," Computing Surveys, Vol. 8(3), pp. 331-353 (Sept. 1976).

3. W. E. Howden, "An Evaluation of the Effectiveness of Symbolic Testing," pp. 300-313, in Tutorial: Software Testing and Validation Techniques, ed. E. Miller & W. E. Howden, IEEE (EH0138-8 or UCSD CS Report 16) (Mar. 1977).

4. W. E. Howden, "Symbolic Testing and the DISSECT Symbolic Evaluation System," IEEE Transactions on Software Engineering, Vol. 3(4), pp. 266-278 (Jul. 1977).

5. J. C. King, "Proving Programs to be Correct," IEEE Transactions on Computers, Vol. 20(11), pp. 1331-1336 (Nov. 1971).

6. J. C. King, "Symbolic Evaluation and Program Testing," Communications of the ACM, Vol. 19(7), pp. 385-394 (Jul. 1976).

BIBLIOGRAPHY ON PROGRAMMING ENVIRONMENTS

1. C. N. Alberga, A. L. Brown, G. B. Leeman, M. Mikelsons, and M. N. Wegman, "A Program Development Tool," pp. 92, in Conference Record of the Eighth Annual ACM Symposium on Principles of Programming Languages (Jan. 1981).

2. R. M. Balzer, "EXDAMS-Executable Debugging and Monitoring System," pp. 567, in AFIPS Proceedings (SJCC) (1969).

3. M. A. Branstad, W. R. Adrion, W. Howden, L. Osterweil, T. Standish, and M. Zelkowitz, "NBS Workshop Report on Programming Environments," Software Engineering Notes, Vol. 6(4), pp. 1-51 (Aug. 1981).

4. J. N. Buxton, "An Informal Bibliography on Programming Support Environments," SIGPLAN Notices, Vol. 15(12), p. 17 (Dec. 1980).

5. P. M. Cashman and A. W. Holt, "A Communications Oriented Approach to Structuring the Software Maintenance Environment," Software Engineering Notes, Vol. 5(1), p. 4 (Jan. 1980).

6. D. Chamberlin, "Janus: An Interactive System for Document Preparation," SIGPLAN Notices, p. 82 (Jun. 1981).

7. P. J. Denning, " "Smart Editors", ACM President's Letter," Communications of the ACM, Vol. 24(8), p. 491 (Aug. 1981).

8. T. A. Dolotta, R. C. Haight, and J. R. Mashey, "The Programmer's Workbench," Bell System Technical Journal (Jul.-Aug. 1978).

9. D. C. Engelbart and W. K. English, "A Research Center for Augmenting Human Intellect," in AFIPS Proceedings (FJCC) (1968).

10. R. E. Fairley, "Ada Debugging and Testing Support Environments," SIGPLAN Notices, Vol. 15(11), p. 16 (Nov. 1980).

11. D. A. Fisher, Design Issues for Ada Program Support Environments: A Catalogue of Issues, Science Applications, Inc., McLean, Va. (Oct. 1980).

12. C. M. Geschke, J. H. Morris, and E. H. Satherthwaite, "Early Experience with MESA," Communications of the ACM (Aug. 1977).

13. B. W. Kernighan and J. R. Mashey, "The UNIX Programming Environment," IEEE Computer, Vol. 14(4), p. 12 (Apr. 1981).

14. B. W. Kernighan and P. J. Plauger, in Software Tools, Addison-Wesley (1976).

15. T. E. Kurtz, "BASIC," SIGPLAN Notices (Aug. 1978).

16. Y. Mano, K. Omaki, and K. Torii, "An Intelligent Multi-display Terminal System -- Towards a Better Programming Environment," Software Engineering Notes, Vol. 6(2), p. 8 (Apr. 1981).

17. M. Mikelsons and M. N. Wegman, PDE1L: The PL1L Program Development Environment Principles of Operation, IBM Research Reports.

18. R. E. Millstein, "The National Software Works: A Distributed Processing System," pp. 44, in Proceedings of the ACM 77 Annual

Conference (Sept. 1977).

19. P. Neumann and S. Gerhart, "Reportage on the 5th ICSE in San Diego," _Software Engineering Notes_, Vol. 6(2), p. 5 (Apr. 1981).

20. D. Prentice, "An Analysis of Software Development Environments," _Software Engineering Notes_, Vol. 6(5), p. 19 (Oct. 1981).

21. J. Ramanathan, "Modeling of Problem Domains for Driving Program Development Systems," pp. 28, in _Conference Record of the Eighth Annual ACM Symposium on Principles of Programming Languages_ (Jan. 1981).

22. B. Reid, "A High Level Approach to Document Formatting," pp. 24, in _Proceedings of the Seventh Annual Symposium on Principles of Programming Languages_ (Jan. 1980).

23. E. Sandewall, "Programming in the Interactive Environment: The LISP Experience," _ACM Computing Surveys_, Vol. 10(1), p. 35 (Mar. 1978).

24. E. Shapiro, G. Collins, L. Johnson, and J. Ruttenberg, "PASES: A Programming Environment for Pascal," _SIGPLAN Notices_, Vol. 16(8), p. 50 (Aug. 1981).

25. T. Teitelbaum and T. Reps, "The Cornell Program Synthesizer: A Syntax-Directed Programming Environment," _Communications of the ACM_, Vol. 24(9), p. 563 (Sept. 1981).

26. W. Teitelman, _A Display-Oriented Programmer's Assistant_, Xerox PARC (Mar. 1977).

27. W. Teitelman, "The INTERLISP Programming Environment," IEEE Computer, Vol. 14(4), p. 25 (Apr. 1981).

28. P. vanderLinden and L. K. Nicholson, "Macro Facilities in the Ada Environment," SIGPLAN Notices, Vol. 16(8), p. 67 (Aug. 1981).

29. N. R. Wasner, "A FORTRAN Preprocessor for the Large Programming Environment," SIGPLAN Notices, Vol. 15(12), p. 92 (Dec. 1980).

30. A. I. Wasserman, "Guest Editor's Introducion: Automated Development Environments," IEEE Computer, Vol. 14(4), p. 7 (Apr. 1981).

31. P. Wegner, "The Ada Language and Environment," Software Engineering Notes, Vol. 5(2), p. 8 (Apr. 1980).

32. "UNIX Issue," Bell System Technical Journal (Jul.-Aug. 1978).

33. "(3 papers)," pp. 34-62, in Proceedings of the 5th Annual Conference on Software Engineering (1981).

34. in STONEMAN (Feb. 1980).

BIBLIOGRAPHY ON SOFTWARE PROTOTYPING

1. T. Gilb, "Evolutionary Development," Software Engineering Notes, Vol. 6(2), p. 17 (Apr. 1981).

2. G. R. Gladden, "Stop the Life-Cycle, I Want to Get Off," Software Engineering Notes, Vol. 7(2), p. 35 (Apr. 1982).

3. S. L. Squires, M. Zelkowitz, and M. Branstad, "Rapid Prototyping Workshop: An Overview," Software Engineering Notes, Vol. 7(3), p. 14 (Jul. 1982).

4. A. I. Wassermann and S. Gutz, "The Future of Programming," Communications of the ACM, Vol. 23(3), p. 196 (Mar. 1982).

5. S. L. Squires, M. Branstad, and M. Zelkowitz, "Special Issue on Rapid Prototyping: Working Papers from the ACM SIGSOFT Rapid Prototyping Workshop," Software Engineering Notes, Vol. 7(5), p. 1-185 ACM Order Number: 592831 (Dec. 1982).

BIBLIOGRAPHY ON FUNCTIONAL LANGUAGES

1. D. P. Friedman and D. S. Wise, "Aspects of Applicative Programming for Parallel Processing," IEEE Transactions on Computers, Vol. C-27(4), pp. 289-296 (April 1978).

2. D. P. Friedman and D. S. Wise, "The Impact of Applicative Programming on Multiprocessing," Proceedings of the 1976 International Conference on Parallel Processing, pp. 263-272 (August 1976).

3. S. A. Ward, "Functional Domains of Applicative Languages," M.I.T. Technical Report MIT/LCS/TR-136 (September 1974).

4. T. Y Feng, "Data Manipulating Functions in Parallel Processors and their Implementations," IEEE Transactions on Computers, Vol. C-23, pp. 309-318.

5. (author n.a.), Proceedings of the ACM Conference on Functional Programming Languages and Computer Architecture (October 1981).

6. Z. Manna and A. Pnueli, "Formalization of Properties of Functional Programs," Journal of the ACM, Vol. 17(3), pp. 555-569 (July 1970).

7. A. Chiarini, "On FP Languages Combining Forms," Technical Report, Comitato Nazionale per L'Energia Nucleare (April 1980).

8. J. C. Peterson and W. D. Murray, "Parallel Computer Architecture Employing Functional Programming Systems," Proceedings of the

International Workshop on High-Level Langauge Computer Architecture, pp. 190-195 (May 1980).

9. J. J. Martin, "Informal Description of Typed Functional Programming Systems," Technical Report CS81015-R, Virginia Polytechnic Institute and State University, Blacksburg, Virginia (September 1981).

10. J. Minne, "A More-Defined Semantics for FFP," UCI Dataflow Architecture Project Note 41, University of California at Irvine (April 1980).

11. K. P. Gostelow, "A General Typing Scheme," UCI Dataflow Architecture Project Note 49, University of California at Irvine (March 1980).

12. J. Backus, "Can Programming be Liberated from the Von Neumann Style? A Functional Style and its Algebra of Programs," _Communications of the ACM_, Vol. 21(8), pp. 613-641 (August 1978).

13. M. P. Georgeff, I. Fris, and J. Kautsky, "The Effect of Operators on Parsing and Evaluation in APL," _Computer Languages_, Vol. 6(2), pp. 67-78 (1981).

14. E. V. Glivenko, T. M. Zhukova, and S. V. Shevchenko, "Algorithm Paralleling and a Multiprocessor Machine of Functional-Statement Type," _Programming and Computer Software_, Vol. 6(5), pp. 249-253 (September-October 1980).

15. M. Nagata, T. Akiyama, and Y. Fujikake, "An Interactive Supporting System for Functional Recursive Programming," _Information_

Processing 80: Proceedings of the IFIP Congress 80, Tokyo, Japan, pp. 263-268, North-Holland (October 1980).

16. H. D. Mills, "Function Semantics for Sequential Programs," Information Processing 80: Proceedings of the IFIP Congress 80, Tokyo, Japan, pp. 241-250, North-Holland (October 1980).

17. L. A. Stabile, "FP and its Use as a Command Language," Proceedings of Distributed Computing: COMPCON 80, 21st IEEE Computer Society International Conference, pp. 301-306 (September 1980).

18. H. B. Hunt, III and D. J. Rosenkrantz, "The Complexity of Recursive Schemes and Recursive Programming Languages," Proceedings of the 21st Annual Symposium on Foundations of Computer Science, pp. 152-160 (October 1980).

19. J.-C. Raoult and J. Vuillemin, "Operational and Semantic Equivalence Between Recursive Programs," Journal of the ACM, Vol. 27(4), pp. 772-796 (October 1980).

20. Arvind, "Decomposing a Program for Multiple Processor Systems," Proceedings of the 1980 International Conference on Parallel Processing, pp. 7-14 (August 1980).

BIBLIOGRAPHY ON VERIFICATION

1. R. B. Anderson, in *Proving Programs Correct*, Wiley, N. Y. (1979).

2. T. A. Anderson, "Provably Safe Programs," *Technical Report, Computing Laboratory, University of Newcastle upon Tyne* (1976).

3. T. Anderson and R. W. Witty, "Safe Programming," *Bit*, Vol. 18(1), pp. 1-8 (1978).

4. G. R. Andrews, "Parallel Programs: Proofs Principles and Practice," *Communications of the ACM*, Vol. 24(3), p. 140 (March 1981).

5. G. R. Andrews and R. P. Reitman, "An Axiomatic Approach to Information Flow in Programs," *ACM Transactions on Programming Languages and Systems*, Vol. 2(1), p. 56 (Jan 1980).

6. K. R. Apt, "Recursive Assertions and Parallel Programs," (*submitted*).

7. K. R. Apt, "Ten Years of Hoare's Logic: A Survey - Part I," *ACM Transactions on Programming Languages and Systems*, Vol. 3(4), p. 431 (October 1981).

8. K. R. Apt, W. P. DeRoever, and N. Francez, "Weakest Precondition Semantics for Communicating Processes," (*to appear*).

9. K. R. Apt, N. Francez, and W. P. DeRoever, "A Proof System for Communicating Sequential Processes," *ACM Transactions on Programming Languages and Systems*, Vol. 2(3), pp. 359-385 (July 1980).

10. E. A. Ashcroft, "Proving Assertions about Parallel Programs," Journal of Computer and System Sciences, Vol. 10, pp. 110-135 (1975).

11. E. A. Ashcroft and Z. Manna, "Formalization of Properties of Parallel Programs," Machine Intelligence, Vol. 6, p. 17 (1971).

12. E. A. Ashcroft and W. W. Wadge, "Intermittent Assertion Proofs in Lucid," Information Processing 77 IFIP Congress Series, Vol. 7, p. 723, North Holland (1977).

13. E. A. Ashcroft and W. W. Wadge, "Lucid: A Nonprocedural Language with Iteration," Communications of the ACM, Vol. 20(7), pp. 519-526 (July 1977).

14. E. A. Ashcroft and W. W. Wadge, "Lucid: A Formal System for Writing and Proving Programs," SIAM Journal of Computing, Vol. 5(3), pp. 336-354 (September 1976).

15. E. A. Ashcroft and W. W. Wadge, "Lucid: Scope Structures and Defined Functions," Technical Report CS-76-22, Computer Science Department, University of Waterloo.

16. E. Ashcroft and W. Wadge, "Some Misconceptions About Lucid," SIGPLAN Notices, Vol. 15(10), pp. 15-26 (October 1980).

17. R. J. R. Back, "On Correct Refinement of Programs," Journal of Computer and System Sciences, Vol. 23(1), pp. 49-68 (August 1981).

18. J. L. Baer, G. Gardarin, C. Girault, and G. Roucairol, "The 2-Step Commitment Protocol-Modeling, Specification and Proof Methodology," Proceedings of the IEEE 5th International Conference on

Software Engineering, p. 363 (1981).

19. M. Ben-Ari, Z. Manna, and A. Pnueli, "The Temporal Logic of Branching Time," pp. 164-176, in Eighth Annual ACM Symposium on Principles of Programming Languages (1981).

20. E. Best, "Proof of a Concurrent Program Finding Euler Paths," pp. 142, in Mathematical Foundations of Computer Science 1980--Lecture Notes in Computer Science (1980).

21. W. Bibel, "A Theoretical Basis for the Systematic Proof Method," pp. 154, in Mathematical Foundations of Computer Science 1980--Lecture Notes in Computer Science (1980).

22. R. S. Boyer and J. S. Moore, "Proving Theorems About LISP Programs," Journal of the ACM, Vol. 22(1), p. 48 (1975).

23. W. H. Burge, "Proving the Correctness of a Compiler," IBM Research Report RC2111 (June 1969).

24. R. Burstall, "Proving Properties of Programs by Structural Induction," Computer Journal, Vol. 12(1), p. 41 (February 1969).

25. R. M. Burstall, "Algebraic Description of Programs with Assertions, Verification and Simulation," ACM Conference on Proving Assertions About Programs, p. 7 (1972).

26. R. M. Burstall, "Some Techniques for Proving Program Correctness of Programs Which Alter Data Structure," Machine Intelligence, Vol. 7, p. 23 (1972).

27. R. Burstall, "Program Proving as Hand Simulation with a Little Induction," pp. 308-312, in Information Processing 74, North-Holland, Amsterdam (1974).

28. R. M. Burstall and P. J. Landin, "Programs and their Proofs: an Algebraic Approach," Machine Intelligence, Vol. 4 (1969).

29. R. M. Burstall and J. W. Thatcher, "The Algebraic Theory of Recursive Program Schemes," pp. 126, in Category Theory Applied to Communication and Control, Springer-Verlag (1975).

30. J. M. Cadiou and J. J. Levy, "Mechanical Proofs About Parallel Processes," Proceedings of the 14th Annual IEEE Symposium on Switching and Automata Theory, p. 34 (1973).

31. R. Cartwright and D. Oppen, "Unrestricted Procedure Calls in Hoare'S Logic," Proceedings of the ACM Symposium on Principles of Programming Languages, p. 131 (January 1978).

32. M. A. Casanova and P. A. Bernstein, "A Formal System for Reasoning About Programs Accessing a Relational Database," ACM Transactions on Programming Languages and Systems, Vol. 2(3), p. 386 (July 1980).

33. K. M. Chandy and J. Misra, "An Axiomatic Proof Technique for Networks of Communicating Processes," Technical Report TR-98, Department of Computer Science, University of Texas at Austin (1979).

34. K. M. Chandy and J. Misra, "Deadlock Absence Proofs for Networks of Communicating Processes," Information Processing Letters, Vol. 9(4), pp. 185-189 (November 1979).

35. Z. C. Chen and C. A. R. Hoare, "Partial Correctness of Communicating Sequential Processes," Proceedings: The 2nd International Conferrence on Distributed Computing Systems, p. 1, IEEE (1981).

36. E. M. Clarke, Jr., "Programming Language Constructs for which it is Impossible to Obtain Good Hoare Axiom Systems," Journal of the ACM, Vol. 26(1), p. 129 (January 1979).

37. E. M. Clarke, Jr., "Synthesis of Resource Invariants for Concurrent Programs," ACM Transactions on Programming Languages and Systems, Vol. 2(3), p. 338 (July 1980).

38. M. Clint, "Program Proving: Coroutines," Acta Informatica, Vol. 2, p. 50 (1973).

39. M. Clint and C. A. R. Hoare, "Program Proving: Jumps and Functions," Acta Informatica, Vol. 1, p. 214 (1972).

40. S. A. Cook, "Axiomatic and Interpretive Semantics for an ALGOL Fragment," Technical Report TR79, Computer Science Department, University of Toronto (1975).

41. S. A. Cook, "Soundness and Completeness of an Axiomatic System for Program Verification," SIAM Journal of Computing, Vol. 7, p. 70 (February 1978).

42. D. C. Cooper, "Program Scheme Equivalences and Second Order Logic," Machine Intelligence, Vol. 4, p. 3 (1969).

43. M. M. Cutler, "Proving Properties of Simulation Programs for System Verification and Validation," Proceedings of the 1979 Summer Computer Simulation Conference, p. 610, IEEE, AFIPS (1979).

44. M. M. Cutler, "Validation Proofs of Discrete Event Simulation Programs," *Proceedings of the 1980 Summer Computer Simulation Conference*, p. 240, Society for Computer Simulation Ifips Press, Arlington, Va. (1980).

45. O. J. Dahl, "An Approach to Correctness Proofs for Semicoroutines," pp. 116, in *Programming Methodology*, ed. D. Gries, Springer-Verlag, New York (1978).

46. J. L. Darlington, "Automatic Theorem Proving with Equality Substitution and Mathematical Induction," *Machine Intelligence*, Vol. 3, p. 113 (1968).

47. J. W. DeBakker, "Axiomatics of Simple Assignment Statements," MR.94, Mathematisch Centrum, Amsterdam (June 1968).

48. J. W. DeBakker, "Flow of Control in the Proof Theory of Structured Programming," *16th Annual Symposium on the Foundations of Computer Science*, p. 29, IEEE Computer Society (1975).

49. J. W. DeBakker, *Mathematical Theory of Program Correctness*, Prentice Hall, Englewood Cliffs, N.J. (1980).

50. J. W. DeBakker, "Recursive Programs as Predicate Transformers," *Proceedings of the IFIP Conference on Formal Specification of Programming Constructs*, p. 7.1 (1977).

51. J. W. DeBakker, "Semantics and Foundations of Program Proving," *Information Processing 77 IFIP Congress Series*, Vol. 7, p. 279, North Holland (1977).

52. J. W. DeBakker, "Semantics of Programming Languages," in _Advances in Information Systems Science_, ed. J. T. Tou, Plenum Press, New York (1969).

53. J. W. DeBakker and L. G. L. T. Meertens, "On the Completeness of the Inductive Assertion Method," _Journal of Computer and System Sciences_, Vol. 11(3), p. 323 (December 1975).

54. R. A. DeMillo, R. J. Lipton, and A. J. Perlis, "Social Processesa and Proofs of Theorems and Programs," _Communications of the ACM_, Vol. 22(5), pp. 271-280 (May 1979).

55. W. P. DeRoever, "Dijkstra's Predicate Transformer, Nondeterminism, Recursion and Termination," pp. 472, in _Lecture Notes in Computer Science_: _Foundations of Computer Science_ (1976).

56. T. J. Dekker, "Correctness Proof and Machine Arithmetic," pp. 31, in _Performance Evaluation of Numerical Software_ (_IFIP_), ed. L. D. Fosdick, North Holland, Amsterdam (1978).

57. N. Dershowitz and Z. Manna, "Proving Termination with Multiset Orderings," _Communications of the ACM_, Vol. 22(8), p. 465 (August 1979).

58. S. C. Dewhurst, "An Equivalence Result for Temporal Logic," _SIG-PLAN Notices_, Vol. 16(2), pp. 53-55 (Feb. 1981).

59. E. W. Dijkstra, "A Constructive Approach to the Problem of Program Correctness," _BIT_, Vol. 8(3), p. 174 (1968).

60. E. W. Dijkstra, "A Correctness Proof for Communicating Processes - a Small Exercise," _EWD-607 Burroughs_, _Nuenen Netherlands_ (1977).

61. E. W. Dijkstra, "A Simple Axiomatic Basis for Programming Language Constructs," Technological University Eindhoven EWD372 (1973).

62. E. W. Dijkstra, "Finding the Correctness Proof of a Concurrent Program," pp. 31, in Mathematical Foundations of Computer Science - Lecture Notes in Computer Science (1978).

63. E. W. Dijkstra, "Notes on Structured Programming," TH Rept. 70-WSK-03, Technische Hogeschool, Eindhoven, the Netherlands (Apr 1970).

64. E. W. Dijkstra, "On a Political Pamphlet from the Middle Ages," ACM SIGSOFT, Vol. 3(2), pp. 14-16 (April 1978).

65. A. G. Duncan and L. Yelowitz, "Studies in Abstract-Concrete Mappings in Proving Algorithms Correct," pp. 218, in Automata, Languages and Programming - Lecture Notes in Computer Science (1979).

66. K. N. Efimkin and I. B. Zadykhailo, "Verification of Programs Written in Symbolic Processor Language," Programming and Computer Software, Vol. 6(2), pp. 103-110 (March-April 1980).

67. H. D. Ehrich and U. Lipeck, "Proving Programs Correct - 2 Alternative Approaches," Information Processing 80 - IFIP Congress Series, Vol. 8, p. 83 (1980).

68. H. D. Ehrich and U. Lipeck, "Proving Implementations Correct--Two Alternative Approaches," Information Processing 80: Proceedings of the IFIP Congress 80, Tokyo, Japan, pp. 83-88, North-Holland (October 1980).

69. B. Elspas, "Specification and Proof of Consistency for Verification Condition Generators," _Proceedings: Specifications of Reliable Software_, p. 212, IEEE (1979).

70. B. Elspas, B. Green, K. N. Levitt, and R. J. Waldinger, in _Research in Interactive Program Proving Techniques_, SRI, Menlo Park, California (May 1972).

71. B. Elspas, K. N. Levitt, R. J. Waldinger, and A. Waksman, "An Assessment of Techniques for Proving Program Correctness ," _ACM Computing Surveys_, Vol. 4(2), pp. 97-147 (June 1972).

72. E. A. Emerson and E. M. Clarke, "Characterizing Correctness Properties of Parallel Programs Using Fixpoints," _Automata, Languages and Programming - Lecture Notes in Computer Science_, Vol. 85, p. 169 (1980).

73. G. W. Ernst, "Rules of Inference for Procedure Calls," _Acta Informatica_, Vol. 8, p. 145 (1977).

74. L. Flon and N. Suzuki, "Consistent and Complete Proof-Rules for Total Correctness of Parallel Programs," _19th Annual Symposium on Foundations of Computer Science_, p. 184, IEEE Technical Committee on the Mathematical Foundations of Computing (1978).

75. R. W. Floyd, "Assigning Meaning to Programs," _Proceedings of the Symposium in Applied Mathematics_, pp. 19-32, American Mathematical Society (1967).

76. N. Francez, C. A. R. Hoare, D. J. Lehmann, and W. P. DeRoever, "Semantics of Nondeterminism Concurrency and Communication,"

Journal of Computer and System Sciences, Vol. 19, p. 290 (1979).

77. N. Francez and A. Pnueli, "A Proof Method for Cyclic Programs," *Acta Informatica*, Vol. 9 (1978).

78. D. Gabbay, A. Pnueli, S. Shelah, and J. Stavi, "On the Temporal Analysis of Fairness," in *Seventh Annual ACM Symposium on Principles of Programming Languages* (1980).

79. M. Geller, "Test Data as an Aid in Proving Program Correctness," *Communications of the ACM*, Vol. 21(5), p. 368 (May 1978).

80. S. L. Gerhart, "Correctness Preserving Program Transformations," *Proceedings of the 2nd ACM Symposium on Principles of Programming Languages*, p. 54 (January 1975).

81. S. L. Gerhart, "Limitations of Proving and Testing," Validation Methods for Fault-Tolerant Avionics and Control Systems: Working Group Meeting I, NASA Conference Publication, NASA Langley Research Center, Hampton, Virginia (1979).

82. S. L. Gerhart, "Program Validation," pp. 66-108, in *Computing System reliability*, ed. B. Randell, Cambridge Press, Cambridge, England (1979).

83. S. L. Gerhart, D. R. Musser, and D. H. Thompson, et al., "An Overview of AFFIRM: A Specification and Verification System," *Information Processing 80: Proceedings of the IFIP Congress 80, Tokyo, Japan*, pp. 343-347, North-Holland (October 1980).

84. S. Gerhart and L. Yelowitz, "Observations on Fallibility in Applications of Modern Programming Methodologies," *IEEE Transactions on*

Software Engineering, Vol. SE-2(3), p. 195 (September 1976).

85. C. Ghezzi and M. Jazayeri, "Syntax Directed Symbolic Execution," Proceedings of the Computer Software and Applications Conference, pp. 539-545 (October 1980).

86. D. I. Good, "Toward a Man-Machine System for Proving Program Correctness," Technical Report TSN-11, University of Texas (June 1970).

87. D. I. Good, R. M. Cohen, and J. Keeton-Williams, "Principles of Proving Concurrent Programs in Gypsy," Proceedings of the 6th ACM Symposium on Principles of Programming Languages, pp. 42-52 (January 1979).

88. D. I. Good and L. C. Ragland, "Nucleus--A Language for Provable Programs," pp. 93, in Program Test Methods, ed. W. Hetzel, Prentice-Hall, Englewood Cliffs, N.J. (1973).

89. I. Greif and A. R. Meyer, "Specifying the Semantics of While Programs: A Tutorial and Critique of a Paper by Hoare and Laurer," ACM Transactions on Programming Languages and Systems, Vol. 3(4), p. 484 (October 1981).

90. D. Gries, "An Exercise in Proving Parallel Programs Correct," Communications of the ACM, Vol. 20(12), p. 921 (December 1977).

91. D. Gries, "An Exercise in Proving Properties of Parallel Programs," Lecture Notes, Technical University of Munich.

92. D. Gries, "An Illustration of Current Ideas on the Derivation of Correctness Proofs and Correct Programs," IEEE Transactions on

Software Engineering, Vol. 2, p. 238 (December 1976).

93. D. Gries, "Educating the Programmer: Notation, Proofs, and the Development Process," Proceedings of IFIP, pp. 935-944 (1980).

94. D. Gries and G. Levin, "Assignment and Procedure Call Proof Rules," ACM Transactions on Programming Languages and Systems, Vol. 2(4), p. 564 (Oct 1980).

95. D. Gries and S. Owicki, "Verifying Properties of Parallel Programs: An Axiomatic Approach," Communications of the ACM, Vol. 19, p. 279 (May 1976).

96. J. V. Guttag, in Proof Rules for the Programming Language EUCLID, University of Southern California Information Science Institute (May 1977).

97. J. V. Guttag, J. J. Horning, and R. L. London, "A Proof Rule for EUCLID Procedures," Technical Report IS/RR-77-60, University of Southern California Information Science Institute (May 1977).

98. B. T. Hailpern, "A Simple Protocol Whose Proof Isn't," Proceedings: Trends and Applications 1981, p. 71, IEEE (1981).

99. S. Hantler and J. King, "An Introduction to Proving the Correctness of Programs," ACM Computing Surveys, Vol. 8(3), pp. 331-353 (September 1976).

100. C. Hewitt, "PLANNER: A Language for Proving Theorems and Manipulating Models in a Robot," Ph.D. Thesis, M.I.T. Department of Mathematics (January 1971).

101. C. Hewitt, G. Attardi, and H. Lieberman, "Specifying and Proving Properties of Guardians for Distributed Systems," _Semantics of Concurrent Computation - Lecture Notes in Computer Science_, Vol. 70, p. 316 (1979).

102. C. A. R. Hoare, in _Parallel Programming: An Axiomatic Approach_, Stanford University Department of Computer Science (1973).

103. C. A. R. Hoare, "An Axiomatic Basis for Computer Programming," pp. 89, in _Programming Methodology_, ed. D. Gries, Springer-Verlag, New York (1978).

104. C. A. R. Hoare, "Procedures and Parameters: An Axiomatic Approach," _Notes in Mathematics #188 : Symposium on Semantics of Algorithmic Languages_, p. 102 (1971).

105. C. A. R. Hoare, "Program Correctness Proofs," pp. 7, in _Formal Aspects of Computing Science_, Newcastle upon Tyne (September 1974).

106. C. A. R. Hoare, "Proof of a Program: FIND," pp. 101, in _Programming Methodology_, ed. D. Gries, Springer-Verlag, New York (1978).

107. C. A. R. Hoare, "Proof of a Structured Program: Sieve of Erastosthenes," _Computer Journal_, Vol. 15, p. 321 (November 1972).

108. C. A. R. Hoare, "Towards a Theory of Parallel Programming," pp. 202, in _Programming Methodology_, ed. D. Gries, Springer-Verlag, New York (1978).

109. C. A. R. Hoare, "Proof of Correctness of Data Representations," _Acta Informatica_, Vol. 1(4), pp. 271-281 (1972).

110. C. A. R. Hoare and M. Foley, "Proof of a Recursive Program Quicksort," Computer Journal, Vol. 14, p. 391 (November 1971).

111. C. A. R. Hoare and P. E. Lauer, "Consistent and Complementary Formal Theories of the Semantics of Programming Languages," Acta Informatica, Vol. 3, p. 135 (1974).

112. C. A. R. Hoare and N. Wirth, "An Axiomatic Definition of the Programming Language PASCAL," Acta Informatica, Vol. 2, p. 335 (1973).

113. J. H. Howard, "Proving Monitors," Communications of the ACM, Vol. 19(5), p. 273 (May 1976).

114. S. Igarashi, "An Axiomatic Approach to Equivalence Problems of Algorithms with Applications," Ph.D. Thesis, Univaesity of Tokyo (1964).

115. G. Kahn, "A Preliminary Theory for Parallel Programs," Research Report 6, IRIA, France (January 1973).

116. D. M. Kaplan, "Proving Things About Programs," Proceedings of the 4th Annual Princeton Conference on Information Science and Systems (March 1970).

117. S. M. Katz and Z. Manna, in Logical Analysis of Programs, The Weizmann Institute of Science (1974).

118. J. C. King, "Proving Programs to be Correct," IEEE Transactions on Computers, Vol. C-20(11), p. 1331 (November 1971).

119. H. T. Kung and S. W. Song, "Efficient Parallel Garbage Collection System and its Correctness Proof," _18th Annual Symposium on Foundations of Computer Science_, p. 120, IEEE (1977).

120. L. Lamport, "'Sometime' is Sometimes 'not Never'," in _Seventh Annual ACM Symposium on Principles of Programming Languages_ (1980).

121. L. Lamport, "A New Approach to Proving the Correctness of Multiprocess Programs," _ACM Transactions on Programming Languages and Systems_, Vol. 1(1), pp. 84-97 (July 1979).

122. L. Lamport, "On the Proof of Correctness of a Calendar Program," _Communications of the ACM_, Vol. 22(10), p. 554 (Oct 1979).

123. L. Lamport, "Proving the Correctness of Multiprocess Programs," _IEEE Transactions on Software Engineering_, Vol. 3(2), p. 125 (1977).

124. L. Lamport, "The Specification and Proof of Correctness of Interactive Programs," _Lecture Notes in Computer Science_: _Proceedings of Mathematical Studies of Information Processing_, Vol. 75, p. 474 (August 1978).

125. H. C. Lauer, "Correctness in Operating Systems," _Ph.D. Thesis_, Carnegie-Mellon University (1973).

126. P. E. Lauer, "Consistent Formal Theories of the Semantics of Programming Languages," _Technical Report TR25.121_, IBM Laboratories Vienna (1971).

127. S. S. Lavrov, "On Problem Solving on a Computer and Proving Program Correctness," Porkl. Mat. i. Mekh. Nauk(Publ.) USSR, p. 98 (1971).

128. J. A. N. Lee, in Computer Semantics, Van Nostrand Reinhold, New York (1972).

129. G. M. Levin, "A Proof Technique for Communicating Sequential Processes (with an Example)," Technical Report, Computer of Science Department, Cornell University (1979).

130. G. M. Levin, "Proof Rules for Communicating Sequential Processes," Ph.D. Thesis, Department of Computer Science, Cornell University (1980).

131. T. E. Lindquist and R. F. Keller, "Correctness of Programs Written in Language KL-1," IEEE Computer Society's First International Computer Software and Applications Conf., p. 635 (1977).

132. R. J. Lipton, "Reduction: A New Method of Proving Properties of Systems of Processes," Proceedings of the 2nd ACM Symposium on Principles of Programming Languages, p. 78 (1975).

133. B. Liskov, "Abstraction Mechanisms in CLU," Communications of the ACM, Vol. 20(6) (June 1977).

134. R. L. London, "Bibliography on Proving the Correctness of Computer Programs," Machine Intelligence, Vol. 5, pp. 569-580 (1970).

135. R. L. London, "Computer Programs Can be Proved Correct," Proceedings of the 4th Systems Symposium - Formal Systems and Non-Numerical Problem Solving by Computers, p. 281, Springer-

Verlag (November 1968).

136. R. L. London, "Correctness of Two Compilers for a LISP Subset," Stanford Artificial Intelligence Project Memo AIM-151, Computer Science Department, Stanford University, Palo Alto, California (October 1971).

137. R. L. London, "Proof of Algorithms: A New Kind of Certification," Communications of the ACM, Vol. 13(6), p. 371 (June 1970).

138. R. L. London, "Software Reliability Through Proving Programs Correct," Publication 71C 6-C, IEEE Computer Society (March 1971).

139. R. L. London, "The Current State of Proving Programs Correct," Proceedings of the ACM Annual Conference, p. 39 (1972).

140. R. London, "A View of Program Verification," SIGPLAN Notices (June 1975).

141. R. L. London, J. V. Guttag, J. J. Horning, B. W. Lampson, J. G. Mitchell, and G. P. Popek, "Proof Rules for the Programming Language Euclid," Acta Informatica, Vol. 10, p. 1 (1978).

142. R. L. London and J. H. Halton, "Proofs of Algorithms for Asymptotic Series," Computer Science Technical Report 54A, University of Wisconsin (May 1969).

143. D. C. Luckham, D. M. R. Park, and M. S. Paterson, "On Formalized Computer Programs," Journal of Computer and System Sciences, Vol. 4(3), p. 220 (June 1970).

144. Z. Manna, "Mathematical Theory of Partial Correctness," Journal of Computer and System Sciences, Vol. 5(3), p. 239 (June 1971).

145. Z. Manna, "Properties of Programs and the First Order Predicate Calculus," Journal of the ACM, Vol. 16(2), p. 244 (April 1969).

146. Z. Manna, "Second Order Mathematical Theory of Computation," Proceedings of the 2nd Annual ACM Symposium on Theory of Computation, p. 158 (1970).

147. Z. Manna, "The Correctness of Programs," Journal of Computer and System Sciences, Vol. 3(2), p. 119 (May 1969).

148. Z. Manna and J. McCarthy, "Properties of Programs and Partial Function Logic," Machine Intelligence, Vol. 5, p. 27 (1970).

149. Z. Manna, S. Ness, and J. Vuillemin, "Inductive Methods for Proving Properties of Programs," Communications of the ACM, Vol. 16(8), p. 491 (August 1973).

150. Z. Manna and A. Pnueli, "Axiomatic Approach to Total Correctness of Programs," Acta Informatica, Vol. 3, p. 243 (1974).

151. Z. Manna and A. Pnueli, "Formalization of Properties of Recursively Defined Functions," Proceedings of the ACM Symposium on Theory of Computation, p. 201 (1969).

152. Z. Manna and A. Pnueli, "The Validity Problem of the 91 Function," Artificial Intelligence Memo 68, Stanford University (August 1968).

153. Z. Manna and R. J. Waldinger, "Is SOMETIME Sometimes Better than ALWAYS? Intermittent Assertions in Proving Program Correctness," _Communications of the ACM_, Vol. 21(2), p. 159 (February 1978).

154. A. Mazurkiewicz, in _A Complete Set of Assertions on Distributed Systems_, Institute of Computer Science, Polish Academy of Science (1979).

155. J. McCarthy, "A Basis for a Mathematical Theory of Computation," in _Computer Programming and Formal Systems_, ed. D. Hirshberg, North Holland, Amsterdam (1963).

156. J. McCarthy, "A Formal Description of a Subset of ALGOL," _Proceedings of a Conference on Formal Language Description Languages_ (1964).

157. J. McCarthy, "Problems in the Theory of Computability," _Proceedings of the IFIP Congress '65_, Vol. 1, p. 219 (1965).

158. J. McCarthy, "Towards a Mathematical Theory of Computation," _Proceedings of the IFIP Congress '62_, North Holland (1963).

159. J. McCarthy and J. Painter, "Correctness of a Compiler for Arithmetic Expressions," _Mathematical Aspects of Computer Science, Proceedings of Symposium in Applied Mathematics_, Vol. 19, American Mathematical Society (1967).

160. M. C. Mcfarland, "On Proving the Correctness of Optimizing Transformations in a Digital Design Automation System," _Proceedings of the ACM IEEE 18th Design Automation Conference_, p. 90 (1981).

161. F. L. Morris, "Advice on Structuring Compilers and Proving them Correct," Proceedings of the ACM Symposium on Principles of Programming Languages (1973).

162. P. Mosses, "A Constructive Approach to Compiler Correctness," Semantics-Directed Compiler Generation - Lecture Notes in Computer Science, Vol. 94, p. 189 (1980).

163. D. Musser, "Abstract Data Type Specification in the AFFIRM System," IEEE Transactions on Software Engineering, Vol. SE-6(1), pp. 24-32 (January 1980).

164. P. Naur, "Proof of Algorithms by General Snapshots," BIT, Vol. 6, p. 310 (1966).

165. P. G. Neumann, "A Provably Secure Operating System: the System, its Applications, and Proofs," Final Report SRI Project 4332, SRI International, Menlo Park, California (February 1977).

166. M. C. Newey, "Proving Properties of Assembly Language Programs," Information Processing 77 IFIP Congress Series, Vol. 7, p. 795, North Holland (1977).

167. G. Newton, "Proving Properties of Interacting Processes," Acta Informatica, Vol. 4, p. 117 (1975).

168. D. C. Oppen and S. A. Cook, "Proving Assertions About Programs That Manipulate Data Structures," Proceedings of the 7th ACM Symposium on Theory of Computation, p. 107 (1975).

169. S. Owicki, "A Consistent and Complete Deductive System for the Verification of Parallel Programs," Proceedings of the 8th ACM

Symposium on Theory of Computation, p. 73 (1976).

170. S. Owicki, "Axiomatic Proof Techniques for Parallel Programs," Ph.D. Thesis, Cornell University, Ithaca, New York (1975).

171. S. Owicki and D. Gries, "An Axiomatic Proof Technique for Parallel Programs," pp. 130, in *Programming Methodology*, ed. D. Gries, Springer-Verlag, New York (1978).

172. S. Owicki and D. Gries, "Verifying Properties of Parallel Programs: An Axiomatic Approach," *Communications of the ACM*, Vol. 19(5), p. 279 (1976).

173. D. Park, "Fixpoint Induction and Proofs of Program Properties," *Machine Intelligence*, Vol. 5, p. 59 (1969).

174. D. A. Patterson, "An Experiment in High Level Language Microprogramming and Verification," *Communications of the ACM*, Vol. 24(10), pp. 699-709 (October 1981).

175. A. Pnueli, "The Temporal Logic of Programs," pp. 46-57, in *Proceedings of the 18th Annual Symposium on Foundations of Computer Science* (1977).

176. A. Pnueli, "The Temporal Semantics of Concurrent Programs," *Theoretical Computer Science*, Vol. 13(1), p. 45 (Jan 1981).

177. W. Polak, "An Exercise in Automatic Program Verification," *IEEE Transactions on Software Engineering*, Vol. SE-5(5), pp. 453-458 (September 1979).

178. V. R. Pratt, "Semantical Considerations on Floyd-Hoare Logic," Proceedings 17th IEEE Symposium on Foundations of Computer Science, p. 109 (Oct 1976).

179. D. Prawitz, "An Improved Proof Procedure," Theoria, Vol. 26, p. 102 (1960).

180. A. Prior, in Past, Present and Future, Oxford University Press, London (1967).

181. P. Pritchard, "Program Proving - Expression Languages," Information Processing 77 IFIP Congress Series, Vol. 7, p. 727, North Holland Pub. Co., Amsterdam (1977).

182. K. Ramamritham and R. M. Keller, "Specifying and Proving Properties of Sentinel Processes," Proceedings: 5th International Conference on Software Engineering, p. 374, IEEE (1981).

183. R. P. Reitman, "Information Flow in Parallel Programs: An Axiomatic Approach," Ph.D. Th. Cornell U. (Aug 1978).

184. R. P. Reitman and G. R. Andrews, "Certifying Information Flow Properties of Programs: An Axiomatic Approach," Proc. 6th Symposium on Principles of Programming Languages, p. 283 (January 1979).

185. N. Rescher and A. Urquhart, in Temporal Logic, Springer Verlag (1971).

186. J. C. Reynolds, "Programming with Transition Diagrams," pp. 153, in Programming Methodology, ed. D. Gries, Springer-Verlag NY (1978).

187. L. Robinson and K. Levitt, in Proof Techniques for Hierarchically Structured Programs, Stanford Research Institute, Menlo Park, Calif. (1975).

188. M. J. Rochkind, "A Table-Driven Validator," Proceedings of Distributed Computing: COMPCON 80, 21st IEEE Computer Society International Conference, pp. 712-715 (September 1980).

189. W. P. Roever, "Recursive Program Schemes-Semantics and Proof Theory," Mathematical Centre Tracts, Vol. 70, Mathematiche Centrum (1976).

190. B. K. Rosen, "Correctness of Parallel Programs: the Church-Rosser Approach," Theoretical Computer Science, Vol. 2, p. 183 (1976).

191. H. Samet, "Automatically Proving the Correctness of Translations Involving Optimized Code," Ph.D. Th. Stanford Artificial Intelligence Project Memo AIM-259, p. C S Dept. Stanford U., Stanford Calif. (1975).

192. H. Samet, "Proving the Correctness of Heuristically Optimized Code," Communications of the ACM, Vol. 21(7), p. 570 (Jul 1978).

193. J. Schwarz, "Event-Based Reasoning - a System for Proving Correct Termination of Programs," Proc. 3rd International Colloquium on Automation, Languages and Programming, p. 131 (July 1976).

194. M. Shaw, W. Wulf, and R. London, "Abstraction and Verification in ALPHARD: Defining and Specifying Iteration and Generators," Communications of the ACM, Vol. 20(8), pp. 553-563 (August 1977).

195. M. Sintzoff, "Verification of Assertions for Functions," Proceedings IRIA Colloquium on Proving and Improving Programs, p. 11 (1975).

196. R. Sites, "Proving that Computer Programs Terminate Cleanly," TR STAN-CS-74-418 Stanford U. Stanford, Calif. (1974).

197. J. N. Spitzen, K. N. Levitt, and L. Robinson, "An Example of Hierarchical Design and Proof," CACM, Vol. 21(12), p. 1064 (Dec 1978).

198. N. Suzuki, "Verifying Programs by Algebraic and Logical Reductions," Proceedings 1975 International Conf. on Reliable Software, p. 473 (April 1975).

199. J. Szlanko, "Petri Nets for Proving Some Correctness Properties of Parallel Programs," Real Time Programming 1977 - IFAC 1977 Conference Proceedings, p. 75, Internat. Federat. Automat. Control / IFIP (1977).

200. W. D. Tajibnapis, "A Correctness Proof of a Topology Information Maintenance Protocol for a Distributed Computer Network," Communications of the ACM, Vol. 20(7), p. 477 (Jul 1977).

201. J. W. Thatcher, E. G. Wagner, and J. B. Wright, "More on Advice on Structuring Compilers and Proving them Correct," Automata, Languages and Programming - Lecture Notes in Computer Science, Vol. 71, p. 596 (1979).

202. J. W. Thatcher, E. G. Wagner, and J. B. Wright, "More on Advice on Structuring Compilers and Proving them Correct," Semantics-

Directed Compiler Generation - Lecture Notes in Computer Science, Vol. 94, p. 165, Springer-Verlag (1980).

203. R. W. Topor, "A Simple Proof of the Schorr-Waite Garbage Collection Algorithm," *Acta Informatica*.

204. V. F. Turchin, "The Use of Metasystem Transition in Theorem Proving and Program Optimization," *Automata, Languages and Programming - Lecture Notes in Computer Science*, Vol. 85, p. 645, European Assoc. Theroet. C. S. (1980).

205. B. Walker, R. Kemmerer, and G. Popek, "Specification and Verification of the UCLA UNIX Security Kernel," *Communications of the ACM*, Vol. 23(2), pp. 118-131 (February 1980).

206. M. Wand, *Induction, Recursion, and Programming*, North-Holland, New York (1980).

207. A. Wang, "An Axiomatic Basis for Proving Total Correctness of GOTO-programs," *BIT*, Vol. 16, p. 88 (1976).

208. B. Wegbreit, "The Synthesis of Loop Predicates," *Communications of the ACM*, Vol. 17(2), p. 102 (February 1974).

209. B. Wegbreit and J. M. Spitzen, "Proving Properties of Complex Data Structures," *Journal of the ACM*, Vol. 23(2), p. 389 (April 1976).

210. W. Wulf, R. London, and M. Shaw, "An Introduction to the Construction and Verification of ALPHARD Programs," *IEEE Transactions on Software Engineering*, Vol. SE-2, pp. 253-264 (1976).

211. R. T. Yeh, "Verifying Programs by Predicate Transformation," pp. 361-376, in _Software Engineering Techniques, state of the art report_, Infotech International, Maidenhead, England (1977).

BIBLIOGRAPHY ON FAULT TOLERANCE

1. T. Anderson, Provably Safe Programs, University of Newcastle upon Tyne Computing Laboratory (Feb. 1975).

2. T. Anderson and R. Kerr, "Recovery Blocks in Action: A System Supporting High Reliability," Proceedings 2nd International Conference on Software Engineering, p. 447 (Oct. 1976).

3. T. Anderson and J. C. Knight, Practical Software Fault Tolerance for Real-Time Systems, University of Newcastle upon Tyne Computing Laboratory (June 1981).

4. T. Anderson and P. A. Lee, Fault Tolerance: Principles and Practice, Prentice-Hall International, London (1981).

5. T. Anderson and P. A. Lee, "The Provision of Recoverable Interfaces," Digest of Papers FTCS-9, p. 87 (June 1979).

6. T. Anderson, P. A. Lee, and S. K. Shrivastava, "A Model of Recoverability in Multilevel Systems," IEEE Transactions on Software Engineering, Vol. 4(6), p. 486 (Nov. 1978).

7. T. Anderson and B. Randell, Computing Systems Reliability, Cambridge University Press, Cambridge (1979).

8. T. Anderson and R. W. Witty, "Safe Programming," BIT, Vol. 18, p. 1 (1978).

9. D. M. Andrews, "Using Executable Assertions for Testing and Fault Tolerance," Digest of Papers FTCS-9, p. 102 (June 1979).

10. M. M. Astrahan, "System R: Relational Approach to Database Management," ACM Transactions on Database Systems, Vol. 1(2), p. 97 (June 1976).

11. A. Avizienis, "Fault-Tolerant Systems," IEEE Transactions on Computers, Vol. 25(12), p. 1304 (Dec. 1976).

12. A. Avizienis and L. Chen, "On the Implementation of N-Version Programming for Software Fault-Tolerance During Program Execution," Proceedings COMPSAC 77, p. 149 (Nov. 1977).

13. J. F. Bartlett, "A 'NonStop' Operating System," Proceedings 11th Hawaii International Conference on Systems Science, p. 103 (Jan. 1978).

14. E. Best, "Atomicity of Activities," Lecture Notes in Computer Science, Vol. 84, p. 225 (1980).

15. E. Best and F. Cristian, Systematic Detection of Exxception Occurrences, Computing Laboratory University of Newcastle upon Tyne (Apr. 1981).

16. J. P. Black, D. J. Taylor, and D. E. Morgan, "A Case Study in Fault Tolerant Software," Software-Practice and Experience, Vol. 11(2), p. 145 (Feb. 1981).

17. B. R. Borgerson, "A Fail-Softly System for Time-Sharing Use," Digest of Papers FTCS-2, p. 89 (June 1972).

18. B. R. Borgerson, "Spontaneous Reconfiguration in a Fail-Softly Computer Utility," Datafair 73 Conference Papers, p. 326 (Apr. 1973).

19. B. R. Borgerson and R. F. Freitas, "An Analysis of PRIME Using a New Reliability Model," Digest of Papers FTCS-4, p. 2.26 (Jan. 1974).

20. R. Boyd, "Restoral of a Real Time Operating System," Proceedings of 1971 ACM Annual Conference, p. 109 (Aug. 1971).

21. A. Brandwajn and R. Joly, "A Scheme for a Fault-Tolerant Virtual Memory," Information Processing Letters, Vol. 10(2), p. 99 (Mar. 1980).

22. R. H. Campbell, K. H. Horton, and G. G. Belford, "Simulations of a Fault-Tolerant Deadline Mechanism," Digest of Papers FTCS-9, p. 95 (Jun. 1979).

23. K. M. Chandy, "Analytic Models for Rollback and Recovery Strategies in Data Base Systems," IEEE Transactions Software Engineering, Vol. 1(1), p. 100 (Mar. 1975).

24. L. Chen, Improving Software Reliability by N-Version Programming, C. S. Dept. UCLA (Aug. 1978).

25. L. Chen and A. Avizienis, "N-Version Programming: A Fault-Tolerance Approach to Reliability of Software Operation," Digest of Papers FTCS-8, p. 3 (June 1978).

26. J. R. Connet, E. J. Pasternak, and B. D. Wagner, "Software Defences in Real-Time Control Systems," Digest of Papers FTCS-2,

p. 94 (June 1972).

27. F. Cristian, "Exception Handling and Software Fault-Tolerance," Digest of Papers FTCS-10, p. 97 (Oct. 1980).

28. C. T. Davies, "Recovery Semantics for a DB/DC System," ACM 73 Annual Conference, p. 136 (Aug. 1973).

29. D. DeAngelis and J. A. Lauro, "Software Recovery in the Fault-Tolerant Spaceborne Computer," Proceedings FTCS-6, p. 143 (June 1976).

30. P. J. Denning, "Fault Tolerant Operating Systems," Computing Surveys, Vol. 8(4), p. 359 (Dec. 1976).

31. W. R. Elmendorf, "Fault-Tolerant Programming," Digest of Papers 1972 FTCS, p. 79 (June 1972).

32. R. S. Fabry, "Dynamic Verification of Operating Sysem Decisions," Communications of the ACM, Vol. 16(11), p. 659 (Nov. 1973).

33. M. A. Fischler, O. Firschein, and D. L. Drew, "Distinct Software: An Approach to Reliable Computing," Proceedings 2nd USA-Japan Computer Conference, p. 573 (Aug. 1975).

34. E. Gelenbe and D. Derochette, "Performance of Rollback Recovery Systems under Intermittent Failures," Communications of the ACM, Vol. 21(6), p. 493 (June 1978).

35. T. Gilb, "Distinct Software: A Redundancy Technique for Reliable Software," pp. 117, in State of the Art Report on Software Reliability, Infotech, Maidenhead (1977).

36. A. Grnarov, J. Arlatt, and A. Avizienis, "On the Performance of Software Fault-Tolerance Strategies," Digest of Papers FTCS-10, p. 251 (Oct. 1980).

37. H. Hecht, "Fault-Tolerant Software," IEEE Transactions on Reliability, Vol. 28(3), p. 227 (Aug. 1979).

38. H. Hecht, "Fault-Tolerant Software for Real-Time Applications," Computing Surveys, Vol. 8(4), p. 391 (Dec. 1976).

39. J. J. Horning, "A Program Structure for Error Detection and Recovery," Lecture Notes in Computer Science, Vol. 16, p. 171 (1974).

40. K. H. Kim, "An Implementation of a Programmer-Transparent Scheme for Coordinating Concurrent Processes in Recovery," Proceedings of COMPSAC 1980, pp. 615-621 (1980).

41. K. H. Kim, "Approaches to Mechanization of the Conversation Scheme Based on Monitors," IEEE Transactions on Software Engineering, Vol. SE-8(3), pp. 189-197 (May 1982).

42. K. H. Kim, "An Approach to Programmer-Transparent Coordination of Recovering Parallel Processes and its Efficient Implementation Rules," Proceedings International Conference Parallel Processing, pp. 58-68 (Aug. 1978).

43. K. H. Kim, "Strategies for Structured and Fault-Tolerant Design of Recovery Programs," Proceedings COMPSAC 78, p. 651 (Nov. 1978).

44. K. H. Kim and C. V. Ramamoorthy, "Failure-Tolerant Parallel Programming and its Supporting System Architecture," AFIPS Conference

Proceedings 1976 National Computer Conference, Vol. 45, p. 413 (June 1976).

45. H. Kopetz, "Software Redundancy in Real Time Systems," IFIP Congress 74, p. 182 (Aug. 1974).

46. P. A. Lee, "A Reconsideration of the Recovery Block Scheme," Computer Journal, Vol. 21(4), p. 306 (Nov. 1978).

47. P. A. Lee, N. Ghani, and K. Heron, "A Recovery Cache for the PDP-11," IEEE Transactions on Computers, Vol. 29(6), p. 546 (Jun. 1980).

48. R. A. Levin, "Program Structures for Exceptional Condition Handling," Ph.D. Thesis, Carnegie Mellon University (1977).

49. D. B. Lomet, "Process Structuring, Synchronization and Recovery Using Atomic Actions," SIGPLAN Notices, Vol. 12(3), p. 128 (Mar. 1977).

50. D. C. Luckham and W. Polak, "Ada Exception Handling: An Axiomatic Approach," ACM Transactions on Programming Languages and Systems, Vol. 2(2), p. 225 (Apr. 1980).

51. P. M. Melliar-Smith and B. Randell, "Software Reliability: The Role of Programmed Exception Handling," SIGPLAN Notices, Vol. 12(3), p. 95 (Mar. 1977).

52. P. M. Merlin and B. Randell, "Consistent State Restoration in Distributed Systems," Digest of Papers FTCS-8, p. 129 (June 1978).

53. D. E. Morgan and D. J. Taylor, "A Survey of Methods of Achieving Reliable Software," Computer, p. 44 (Feb. 1977).

54. S. M. Ornstein, "Pluribus- A Reliable Multiprocessor," AFIPS Conference Proceedings 1975 NCC, Vol. 44, p. 551 (May 1975).

55. A. Prneli, "Temporal Logic," Theoretical Computer Science, Vol. 13(1) (January 1981).

56. B. Randell, "System Structure for Software Fault Tolerance," pp. 195, in Current Trends in Programming Methodology, ed. R. T. Yeh, Prentice-Hall, Englewood Cliffs, NJ (1977).

57. B. Randell, P. A. Lee, and P. C. Treleaven, Reliable Computing Systems, University of Newcastle upon Tyne Computing Laboratory (May 1977).

58. J. G. Robinson and E. S. Roberts, "Software Fault-Tolerance in the Pluribus," AFIPS Conference Proceedings 1978 National Computer Conference, Vol. 47, p. 563 (June 1978).

59. J. A. Rohr, "STAREX Self-Repair Routines: Software Recovery in the JPL-STAR Computer," Digest of Papers FTCS-3, p. 11 (June 1973).

60. D. L. Russel, "State Restoration in Systems of Communicating Processes," IEEE Transactions Software Engineering, Vol. 6(2), p. 183 (Mar.1980).

61. D. L. Russel, "Process Backup in Producer-Consumer Systems," Proceedings 6th ACM Symposium on Operating Systems Principles, p. 151 (Nov. 1977).

62. D. L. Russel and M. J. Tiedeman, "Multiprocess Recovery Using Conversations," Digest of Papers FTCS-9, p. 106 (June 1979).

63. S. K. Shrivastava, "Structuring Distributed Systems for Recoverability and Crash Resistance," Transactions on Software Engineering, Vol. 7(4), p. 436 (Jul. 1981).

64. S. K. Shrivastava, "Concurrent Pascal with Backward Error Recovery: Implementation," Software-Practice and Experience, Vol. 9(12), p. 1021 (Dec. 1979).

65. S. K. Shrivastava, "Concurrent Pascal with Backward Error Recovery: Language Features and Examples," Software-Practice and Experience, Vol. 9(12), p. 1001 (Dec. 1979).

66. S. K. Shrivastava, "Sequential Pascal with Recovery Blocks," Software-Practice and Experience, Vol. 8(2), p. 177 (Mar. 1978).

67. S. K. Shrivastava and A. A. Akinpelu, "Fault Tolerant Sequential Programming Using Recovery Blocks," Digest of Papers FTCS-8, p. 207 (Jun. 1978).

68. S. K. Shrivastava and J. P. Banatre, "Reliable Resource Allocation Between Unreliable Processes," IEEE Transactions on Software Engineering, Vol. 4(3), p. 230 (May 1978).

69. R. M. Simpson, A Study in the Design of Highly Integrated Systems, University of Newcastle upon Tyne Computing Laboratory (Nov. 1974).

70. D. J. Taylor, D. E. Morgan, and J. P. Black, "Redundancy in Data Structures: Improving Software Fault Tolerance," IEEE Transactions

on *Software Engineering*, Vol. 6(6), p. 585 (Nov. 1980).

71. D. J. Taylor, D. E. Morgan, and J. P. Black, "Redundancy in Data Structures: Some Theoretical Results," *IEEE Transactions on Software Engineering*, Vol. 6(6), p. 595 (Nov. 1980).

72. J. S. M. Verhofstad, *On Multi-Level Recovery: An Approach Using Partially Recoverable Interfaces*, University of Newcastle upon Tyne Computing Laboratory (May 1977).

73. J. S. M. Verhofstad, *Recovery for Multi-Level Data Structures*, University of Newcastle upon Tyne Computing Laboratory (Dec. 1976).

74. M. Ward and J. C. D. Nissen, "Software Security in a Stored Program Controlled Switching System," *International Switching Symposium Record*, p. 570 (June 1972).

75. A. I. Wei, "Application of the Fault-Tolerant Deadline Mechanism to a Satellite On-Board Computer System," *Digest of Papers FTCS-10*, p. 107 (Oct. 1980).

76. W. G. Wood, *Recovery Control of Communicating Processes in a Distributed System*, Computing Laboratory, University of Newcastle upon Tyne (Nov. 1980).

77. O. B. vonLinden, "Computers Can Now Perform Vital Functions Safely," *Railway Gazette International*, Vol. 135(11), p. 1004 (Nov. 1979).

BIBLIOGRAPHY ON AUTOMATIC PROGRAMMING

1. I. A. Budyacherskii, "Method of Program Generating Employing a Database for Automated-Research Programs," _Automated Control and Computer Science_, Vol. 12(3), pp. 72-74 (1978).

2. V. N. Red'ko, "Program Composition and Composition Programming," _Programming and Computer Software_, Vol. 4(5), pp. 303-317 (September-October 1978).

3. N. N. Nepeivoda, "Synthesis of Correct Programs out of Correct Subroutines," _Programming and Computer Software_, Vol. 5(1), pp. 10-19 (January-February 1979).

4. V. P. Gritsai and G. E. Tseitlin, "Some Problems of Automated Structural Parallel Programming," _Cybernetics_, Vol. 15(1), pp. 122-131 (January-February 1979).

5. J. R. Abrial and S. A. Schuman, "Nondeterministic System Specification," pp. 34-50, in _Semantics of Concurrent Computations, Evian, France_, Springer-Verlag, Berlin (July 1979).

6. _Advances in Computers_, Academic Press, London (1977).

7. N. S. Prywes, "Automatic Generation of Computer Programs," pp. 58-127, in _Advances in Computers_, ed. M. C. Yovits, Academic Press, London (1977).

8. Z. Manna and R. Waldinger, "A Deductive Approach to Program Synthesis," ACM Transactions on Programming Languages and Systems, Vol. 2(1), pp. 90-121 (January 1980).

9. (author n.a.), Automatic Programming and its Applications, January 1975-May 1981, NTIS (May 1981).

10. A. J. Blikle, "On the Development of Correct Specified Programs," IEEE Transactions on Software Engineering, Vol. SE-7(5), pp. 519-527 (September 1981).

11. D. McDermott, "The PROLOG Phenomenon," SIGART Newsletter, (72), pp. 16-20 (July 1980).

12. V. A. Mihailyuk, "A Method of Program Generation in Current ES Program Packages," Programming and Computer Software, Vol. 5(6), pp. 396-400 (November-December 1979).

13. E. A. Zhogolev, "Syntax Directed Program Construction," Programming and Computer Software, Vol. 5(6) (November-December 1979).

14. W. Bibel, "Syntax-Directed, Semantics-Supported Program Synthesis," Artificial Intelligence, Vol. 14(3), pp. 243-261 (October 1980).

15. E. H. Tyugu, "Towards Practical Synthesis of Programs," Information Processing 80: Proceedings of the IFIP Congress 80, Tokyo, Japan, pp. 207-219, North-Holland (October 1980).

16. R. B. McKee, "Computer... Program Thyself," Data Management, Vol. 18(10), pp. 36-37 (October 1980).

17. J. McCool, "Programs That Are Better--In Any Language," Datalink, pp. 12-13 (September 8, 1980).

18. E. Orlowska, "Resolution Systems and their Applications, I.," Annals Societatis Mathematicae Polonae, Series IV: Fundamenta Informaticae, Vol. 3(2), pp. 235-267 (1980).

19. R. B. K. Dewar, et al., "The NYU Ada Translator and Interpreter," SIGPLAN Notices, Vol. 15(11), pp. 194-201 (November 1980).

20. K. Okada, K. Futatsugi, and K. Torii, "Reliable Program Derivation in Functional Languages by Applying Jackson's Design Method," 10th International Symposium on Fault-Tolerant Computing, pp. 91-96 (October 1980).

21. L. Baxter, "The Versatility of PROLOG," SIGPLAN Notices, Vol. 15(12), pp. 15-16 (December 1980).

22. R. A. Mueller, "Formalization and Automated Synthesis of Microprograms," MICRO 13: Proceedings of the 13th Annual Microprogramming Workshop, pp. 45-53 (November 1980).

23. A. Mercy and M. Molnar, "The MOZ-ART Technology," Informacio Elektronika, Vol. 15(6), pp. 310-313 (1980).

24. M. B. Trakhtenbrot, "Transformations Predefining a Program," Cybernetics, Vol. 16(2), pp. 221-227 (March-April 1980).

25. B. Kelley, "Interactive Design Simplifies Test-Program Generation," Electronic Design, Vol. 29(4), pp. 169-173 (February 19, 1981).

26. O. N. Perminov, "Mechanical Programming Procedures," Programming and Computer Software, Vol. 6(2), pp. 93-97 (March-April 1980).

27. R. Balzer, "Transformational Implementation: An Example," IEEE Transactions on Software Engineering, Vol. SE-7(1), pp. 3-14 (January 1981).

28. M. Broy and P. Pepper, "Program Development as a Formal Activity," IEEE Transactions on Software Engineering, Vol. SE-7(1), pp. 14-22 (January 1981).

29. E. Deak, "A Transformational Derivation of a Parsing Algorithm in a High-Level Language," IEEE Transactions on Software Engineering, Vol. SE-7(1), pp. 23-31 (January 1981).

30. D. S. Wile, "Type Transformations," IEEE Transactions on Software Engineering, Vol. SE-7(1), pp. 32-39 (January 1981).

31. T. Teitelbaum and T. Reps, "The Cornell Program Synthesizer: A Syntax-Directed Programming Environment," Communications of the ACM, Vol. 24(9), pp. 563-573 (September 1981).

32. P. Zave, "An Operational Approach to Requirements Specification for Embedded Systems," IEEE Transactions on Software Engineering, Vol. SE-8(3), pp. 250-269 (May 1982).

33. T. Murata, "Synthesis of Decision-Free Concurrent Systems for Prescribed Resources and Performance," IEEE Transactions on Software Engineering, Vol. SE-6(6), pp. 525-530 (November 1980).