Program Debugging with Toolkits

by

Joseph Chiu Leung Wu

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF
THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE
in the School
of
Computing Science

SIMON FRASER UNIVERSITY
September 1987

APPROVAL

Name: Joseph Chiu Leung Wu

Degree: Master of Science

Title of Thesis: Program Debugging with Toolkits

Examining Committee:

Chairperson: Dr. Wo Shun Luk

Senior Supervisor: Dr. Robert D. Cameron

Dr. Joseph G. Peters

Dr. James J. Weinkam

External Examiner: Dr. Lou J. Hafer

Date Approved: September 14, 1987

PARTIAL COPYRIGHT LICENSE

I hereby grant to Simon Fraser University the right to lend my thesis, project or extended essay (the title of which is shown below) to users of the Simon Fraser University Library, and to make partial or single copies only for such users or in response to a request from the library of any other university, or other educational institution, on its own behalf or for one of its users. I further agree that permission for multiple copying of this work for scholarly purposes may be granted by me or the Dean of Graduate Studies. It is understood that copying or publication of this work for financial gain shall not be allowed without my written permission.

Title of Thesis/Project/Extended Essay

PROGRAM DEBUGGING WITH TOOLKITS

Author: _____

(signature)

JOSEPH CHIU LEUNG WU

(name)

Sept. 15/87

(date)

ABSTRACT

Traditional debugging tools provide primitive program monitoring facilities with which general debugging can be carried out. These tools thus can be characterized as *general-purpose*. Debugging with general-purpose tools entails the understanding of the dynamic behavior of programs through data captured during execution. The voluminous amount of information flow between the user and the debugging system can easily overwhelm one's administrative capacity and patience.

Debugging can be liberated from the ad hoc style of program tracing through the use of *special-purpose* tools. Each special-purpose tool is designed to cope with a class of programming errors and incorporates an effective procedure to assist the detection and diagnosis of such errors. Unimportant clerical details can be relegated to these tools. thereby allowing efforts to be concentrated on the problem-solving aspect of debugging. A debugging facility composed of a collection of special-purpose tools is called a *toolkit system*. This thesis considers issues in the design and implementation of toolkit systems. Example toolkit components are described and an overall evaluation of the toolkit approach is presented.

ACKNOWLEDGEMENTS

I am deeply indebted to my supervisor, Rob Cameron, not only for his technical contributions to this thesis, but also for his insights to many facets of software development. He has been a constant source of inspiration, without which this thesis would not have been possible. His patience and encouragement during the struggling periods are very much appreciated.

Gratitudes are extended to other members of my examining committee, Lou Hafer, Joe Peters, and Jay Weinkam. They have made many valuable comments about the thesis.

I am thankful to my fellow grad students, particularly the gang in Radandt Hall, for bringing life to an otherwise dull working environment. Special thanks to Brent Johnston, Ed Merks, and Bob Neville for their effort in proofreading the manuscripts.

I would like to pay tribute to a special friend, Doris Anderson. She has accompanied and guided me through the early years of my endeavor to Canada. She has also taught me most of the English that I know today. Her means of instruction, including word games such as Scrabble, have been both educational and fun. I am most grateful for this skill.

Finally, I acknowledge the unfailing support of my family.

TABLE OF CONTENTS

APPROVAL ii
ABSTRACT iii
ACKNOWLEDGEMENTS iv
LIST OF TABLES vii
LIST OF FIGURES viii
1. INTRODUCTION 1
1.1. Software Errors 1
1.2. Program Debugging 2
1.3. Current Debugging Tools 4
1.3.1. Traditional Debugging Tools 4
1.3.2. Static Analysis Tools 6
1.3.3. Knowledge-Based Debuggers 7
1.4. Outline 8
1.5. A Note About Typography 9
2. A METHODICAL DESIGN FRAMEWORK 11
2.1. The Toolkit Paradigm 11
2.2. Human Engineering in Design 14
2.3. Taxonomy of Features 17
2.3.1. Semantic Error Detection 18
2.3.2. Consistency Checking 20
2.3.3. Code Auditing 22
2.3.4. Data Structure Analysis 23
2.4. Program Analysis Methods 26
2.5. Implementation 30
2.5.1. Model 30
2.5.2. Program Instrumentation Techniques 32
2.5.3. Efficiency Considerations 38
2.6. Pragmatic Issues 40
3. EXAMPLE TOOLKIT COMPONENTS 42
3.1. Uninitialized Variable Access Detection 42
3.2. Bounded Execution Failure Diagnosis 50
3.3. Parameter Usage Checking 54

4. AN ASSESSMENT 58
5. EPILOGUE 61
APPENDIX A. ALGORITHM FOR DETECTING UNINITIALIZED VARIABLE ACCESS 62
A.1. Exhaustive Monitoring 62
A.2. The Reduction Algorithm 64
A.2.1. Handling Goto Statements 76
A.2.2. Handling Procedure Calls 78
A.2.3. Complete Listing of Algorithm 82
A.2.3.1. The Analyze Procedure 82
A.2.3.2. Auxiliary Procedures 85
APPENDIX B. LISTING OF INSTRUMENTED PROGRAMS 89
B.1. Uninitialized Variable Access Detection 89
B.2. Bounded Execution Failure Diagnosis 92
B.3. Parameter Usage Checking 101
REFERENCES 103
AUTHOR INDEX 110

LIST OF TABLES

Table 2-1:	Dynamic Semantic Errors	20
Table 2-2:	Consistency Checks	21
Table 2-3:	Implementation-Dependent Features	23

LIST OF FIGURES

Figure 1-1:	Systematic Debugging Methods	4
Figure 2-1:	Erroneous Sort Procedure	16
Figure 2-2:	Structure of Debugging System	31
Figure A-1:	Example to Illustrate the Reduction in Monitor Statements	64
Figure A-2:	Syntax of Pascal Statements	69
Figure A-3:	Computing the Analysis State of Statements	71
Figure A-4:	Handling Goto Statements and Procedure Calls	79

CHAPTER 1

INTRODUCTION

Debugging appears to be the single part of the software-production process that programmers seem to abhor the most.

Glenford Myers [51]

1.1. Software Errors

It is a notorious fact that *bugs* have infected every significant piece of software. Studies by Jones [44] indicate that at least one error is introduced for every 67 lines of executable source code produced and that annual expenditure on defect removal exceeds $7.5 billion in the United States alone. Shooman and Bolsky [63] report that correcting an error takes an average of 4.44 man-hours, with the worst case being 35 man-hours. These alarming figures signify an urgent need for practical techniques to improve software reliability.

The current practice for coping with software errors is by means of program testing. Testing is concerned with exposing errors that were previously unknown. A wide variety of techniques, such as functional testing [38], symbolic testing [37], testing by instrumentation [39, 40], and program verification [18, 33, 35], have been proposed and applied in practice with varying degree of success. In all cases, debugging follows in order to eliminate errors discovered.

Another approach is to take preventive measures rather than resort to after-the-fact

cures. The central idea is to adopt a more disciplined attitude toward programming. Methodologies introduced include structured programming [17], zero-defect development [21], and defensive programming. These methodologies have proved to be very effective in reducing the chance of error; nevertheless, the possibility of error still persists. A more ambitious approach aims at eliminating sources of programming errors altogether. Proponents of this approach devote attention to the automatic construction of programs from specifications [16, 29, 48]. Here, the problem is merely shifted to a different level. Naturally, errors will be committed regardless of the notation used, and removing errors from these very-high-level notations is not necessarily any easier than from their conventional counterparts. More sadly, development in automatic programming is still in its infancy, and only toy problems can be tackled successfully.

A conceptual breakthrough that leads to the production of error-free software is unlikely in the foreseeable future. In the meantime, testing and debugging will remain essential to the attainment of quality software. This thesis addresses the software reliability problem from such a viewpoint. We investigate the fundamental issues in constructing practical and effective supporting tools for the elimination of programming errors. We restrict our attention to debugging tools, but, as we shall see, these tools may also be used for testing.

1.2. Program Debugging

Program debugging involves the diagnosis and correction of errors. When an error is initially discovered, only *symptoms*, or external manifestations of the error, are apparent. Common symptoms include abnormal termination, incorrect output, and missing output. Error diagnosis attempts to relate error symptoms to an actual cause.

Once the cause of an error is identified, error correction follows to remove the discrepancy between program behavior and the intended effect. This requires modifications to either the source code or the documentation associated with earlier phases of the development life cycle, depending on where the error was committed. This thesis is primarily concerned with coding errors in transforming a *correct* design into an executable program, so-called *logic errors*.

In considering automated support for debugging, this work will concentrate on error diagnosis tools rather than error correction tools. Automating error correction appears to be generally infeasible with current technology. This is because there are many different ways to achieve the same computational effect and it is most doubtful that the best correction strategy in every situation will be selected by mechanical means, if any strategy can be selected at all. Moreover, error diagnosis and error correction need not be separate activities; pertinent information for correction is often obtained during error diagnosis. In fact, Myers [51] suggests that diagnosis accounts for 95% of overall activity in debugging. The term debugging hereinafter means error diagnosis unless the context requires otherwise.

Error diagnosis is a trial-and-error task often attempted with semi-automated tools. A commonly used method is to naively collect and examine execution information hoping that anomalous conditions leading to the source of error will be recognized. Another prevailing method is backtracking [51], which involves tracing the program in reverse execution order starting from the point of error until the cause is pinpointed. These brute force methods are appealing because they are simple to apply, but the amount of labor required is often intolerable for large programs. Other more systematic methods, namely debugging by induction and debugging by deduction [51], involve deriving hypotheses about the error, which are subsequently subjected to

repeat
analyze available information
devise a hypothesis about the error
verify the hypothesis
until *cause is known*

(a) Debugging by Induction

repeat
list all conceivable causes
eliminate the impossible ones by simple reasoning
verify remaining hypotheses
until *cause is known*

(b) Debugging by Deduction

Figure 1-1: Systematic Debugging Methods

verification. Figure 1.1 outlines the procedure employed. Although using systematic methods will generally improve debugging time, their apparently burdensome nature has been the main impediment to their gaining widespread acceptance. Debugging productivity is directly related to the tools used in that they can influence the debugging style adopted

1.3. Current Debugging Tools

1.3.1. Traditional Debugging Tools

Traditional debugging tools, often called interactive debuggers, provide mechanisms to monitor the run time behavior of programs. Typical facilities offered are the ability to suspend program execution, to trace execution flow, and to examine and modify the execution state. In the early development of interactive debuggers, these facilities were only supported at the object code level [2, 24, 42]. Today, debuggers that operate at the source language level are widespread [7, 9, 19, 26, 31, 60]. Aside from supporting higher level program monitoring facilities, some modern interactive

debuggers [15, 20, 49] take advantage of advanced hardware, such as high-resolution displays and pointing devices, to provide a more convenient user interface. Further improvements to interactive debuggers are directed at extending the current capability for debugging concurrent programs [6, 8, 68] and programs that are written in multiple source languages [9, 14, 43].

The facilities provided by interactive debuggers dictate a tracing style of debugging. Because very little is known about how program monitoring can be used effectively and efficiently, tracing tends to proceed in an ad hoc, brute-force manner. This often results in prolonged debugging time. Systematic debugging methods, outlined in Figure 1-1, may be followed. In these methods, verifying hypotheses is a major step. Using interactive debuggers for this task entails translating each hypothesis into conditions that are known to be true at different points of program execution. Debugging commands are then issued to implement the necessary breakpoints. Upon reaching each breakpoint, the execution state is examined manually to check for violation of the predetermined conditions. This procedure is repeated until the error is properly diagnosed. Although the use of systematic methods will generally improve debugging productivity, the amount of preplanned and coordinated activity appears to be too demanding of the user. Consequently, systematic debugging is seldom attempted.

Interactive debuggers have two major shortcomings. The first lies in the debugging style imposed. Debugging by tracing requires the programmer to cope with and comprehend a vast number of bookkeeping details. This is counterproductive in itself. The other shortcoming is that the program monitoring facilities provided are not suitable for supporting systematic debugging methods. The programmer is thus discouraged from doing so. In our view, if significant debugging productivity is to be achieved, tools must deviate from the tracing style of debugging and gear toward support for systematic debugging.

1.3.2. Static Analysis Tools

Static analysis tools provide useful information for debugging by systematically examining the program source text, but without actually executing the program. Cross-reference generators are familiar examples of static analysis tools, though they are of limited use for debugging. More sophisticated static analyzers are capable of generating diagnostics that are directly related to programming mistakes, using program flow analysis [34, 50] as the underlying methodology. DAVE [27, 52] is a good representative example of such systems. It detects inconsistent usage of variables, such as referencing uninitialized variables and consecutive assignments to the same variable without an intervening reference. LINT [47], a well-known utility for C programs, offers more extensive program checking capabilities, including more strict type checking (than that performed by standard C compilers) and detection of nonportable constructs. MAP [67] takes a novel approach in the presentation of diagnostics: it stores the available information in a database and offers a query language for retrieving the details of interest.

Static analysis tools are attractive for several reasons. First of all, they are easy to operate in that little effort is involved in initiating them to generate diagnostics. This is an important benefit to the user. Secondly, the types of diagnostics produced are more helpful for debugging than the execution state information obtained from interactive debuggers. A final advantage is the thoroughness of checking offered. Instead of focusing on a single execution error, the diagnostics cover all (possible) instances of the same programming mistake, including the ones that lie outside of the particular execution path under consideration. Despite these advantages, static analysis tools have received limited acceptance as debugging aids. This is due partly to the narrow scope of current static analyzers and partly to the inherent theoretical

limitations of static techniques (See Section 2.4). Remedies to these problems must be devised before the techniques of static analysis can be effectively utilized.

1.3.3. Knowledge-Based Debuggers

Knowledge-based debuggers apply techniques from Artificial Intelligence to partly mechanize the debugging process. Typical of such systems are a fixed debugging procedure and an extensible knowledge base. The knowledge base can be defined *a priori* or can be acquired (and even learned) through interaction with an external agent during the operation of the system.

A noteworthy work in this field is Shapiro's PDS system[1] [62]. This system is capable of diagnosing and correcting Prolog programs that contain three general classes of errors: termination with incorrect output, termination with missing output, and apparent nontermination. The system mimics debugging by tracing using an augmented Prolog interpreter for monitoring execution. The knowledge required deals mainly with the input/output behavior of the monitored program and is acquired dynamically through queries presented to the user.

The FALOSY system of Sedlmeyer, *et al.* [61] concentrates on error diagnosis. The system, using a predefined knowledge base, proposes error hypotheses which are subjected to verification. An error hypothesis is a functional model of a program with a built-in defect. A hypothesis is verified by performing some pattern-matching between the proposed functional model and the given program. If pattern-matching succeeds, then the expected defect is confirmed and is reported. Otherwise another

[1]Shapiro's work is concerned with a theoretical framework for debugging. He has also extended the results to program synthesis. Our superficial treatment here can hardly do justice to this significant work. It is recommended reading for interested readers.

hypothesis is generated. The procedure is repeated until exhaustion of hypotheses, in which case system failure is reported.

Other work on knowledge-based debugging has been reported by Adam and Laurent [1], Gupta and Seviora [32], Ruth [59], and Sussman [65].

Knowledge-based debuggers are only at early experimental stages. Very little success has been achieved to date. The difficulties encountered are fundamental and nontrivial. For instance, the issues of what kinds of knowledge are necessary and how they can be effectively captured, represented, and generalized are only vaguely understood. The underlying problem-solving strategy employed by current knowledge-based debuggers, which often involves searching a vast solution space for an appropriate answer, is computationally unacceptable for any realistic undertaking, even with the most modern computer hardware. Practical solutions to these problems are far over the horizon. Although Artificial Intelligence techniques may have an important impact in the long run, more practical alternatives can be investigated in the interim.

1.4. Outline

This thesis is concerned with a methodology for designing advanced debugging systems. From the brief survey on current debugging aids, several desirable characteristics of an advanced system are evident. First of all, the system should directly support a disciplined approach to debugging, through which improved debugging productivity is to be realized. It is essential that in place of low-level facilities, high-level debugging functions that reflect the nature of systematic debugging be provided. Secondly, human factors in tool design should be addressed. A debugging tool, regardless of sophistication, is only useful if it will be accepted by

the user. To this end, simplicity of use is a key point. Finally, in order to be practical, resource requirements of the system must be adequately handled by current computing hardware. This restricts implementation techniques to those whose practicality has been demonstrated. The subject of this work is a design framework which accommodates the abovementioned pragmatic issues within the constraints of current technology.

This thesis is organized as follows. Chapter 2 expounds on a framework for constructing advanced debugging systems. A new approach in structuring debugging systems is first introduced. Design and implementation issues are then addressed. The principles and techniques discussed will be demonstrated in Chapter 3 through examples. An overall evaluation of our approach is given in Chapter 4. Concluding remarks are presented in Chapter 5. Appendix A elaborates on algorithms for detecting uninitialized variable access, which is one of the tool examples to be discussed in Chapter 3.

Our discussion is oriented toward the predominant Algol family of languages. For concreteness, ANSI Standard Pascal [5] has been chosen as the language for illustrating the ideas.

1.5. A Note About Typography

Pascal programs are presented with keywords in lower-case bold and identifiers in entire upper case. For reasons of clarity, underscores are used as word separators within identifiers, even though this is not permitted in Standard Pascal. This style, adopted from the Ada[2] Programming Language Reference Manual [55] and admittedly

---

[2]Ada is a registered trademark of the U.S. Government (Ada Joint Program Office).

not the most pleasing in the aesthetic sense, allows mixing program notations in running text without additional typographical concerns. For example, the sentence

> "The *cardinality* of a set, computed by the function CARDINALITY, is the number of elements in the set."

clearly shows the two roles of "cardinality". It is evident that readability will be reduced if italics are used for identifiers. Using a different font for program notations is also arguably unpleasant.

Algorithms in the appendix are presented in bold and italics to distinguish them from Pascal notations.

CHAPTER 2

A METHODICAL DESIGN FRAMEWORK

. . . [software] engineers balance scientific principles, economic issues, and social concerns in a pragmatic manner when solving problems and developing technological products. . . . Intangibility and lack of physical properties for software limit the number of fundamental guidelines and basic constraints available to shape the design and implementation of a software product.

Richard Fairley [25]

2.1. The Toolkit Paradigm

Designers of traditional debugging tools take the view that debugging can be performed by observing the program dynamics. Hence, mechanisms are provided to examine the program state at various points of execution. Its ramification is that reasoning about programs must be perceived in terms of the underlying execution model. This approach to debugging system design is evidently low-level.

The low-level approach has the merit that general debugging can be carried out with the primitive facilities provided. Traditional debugging tools thus can be characterized as *general-purpose* tools. Achieving generality, however, has greatly compromised usability. Every detail of the entire debugging process must now be attended to. The massive amount of information flow between the user and the debugging system often overwhelms one's administrative capacity and patience. Much of the work involves bookkeeping details that require little intellectual sophistication. An advanced system must remove these unimportant levels of clerical detail. Identifying a new

level of abstraction in the debugging process is thus the paramount issue in high-level debugging system design.

Recall from Section 1.2 that deductive and inductive debugging methods involve deriving and verifying error hypotheses. Deriving hypotheses, which requires knowledge of the problem and programming expertise, is more mentally demanding than the verification part, which is largely laborious and tedious. The verification process can be modeled as procedural abstractions; that is, the concrete operations needed to test a hypothesis are encapsulated to form a single entity and represented conceptually as an atomic operation. By suppressing irrelevant details, efforts can be concentrated on the problem solving aspect of debugging. A system adopting this principle of abstraction is naturally composed of a collection of tools, with each tool implementing a procedural abstraction. Such a high-level system, characterized by its constituent parts, is called a *toolkit system*.

Toolkit systems have a number of important characteristics not typified in conventional systems. As a consequence of the design decision adopted, toolkit systems support a more abstract view of debugging. Each tool can be viewed as an oracle to certain conjectures on program misbehavior. Because low-level operations such as crude examination of program state changes are not provided, debugging liberally without forethought is discouraged. Although an abuse of tools cannot be avoided, systematic debugging is better supported and will more likely be adhered to.

Another distinctive feature of toolkit systems is that each individual tool can only be applied to detect a specific kind of error and is therefore said to be *special-purpose*. Because of its confined scope, effectiveness can be attained by incorporating into the tool knowledge specific to its application domain. Special-

purpose tools also can be made easy to use: because of the built-in knowledge, minimal human guidance is needed to operate them. In contrast, early (and even some existing) interactive debuggers are typically difficult to learn and operate; this often diverts the user to manual methods. From the users' perspective, achieving effectiveness without sacrificing simplicity of use is the most appealing aspect of toolkit systems.

In contrast to conventional monolithic systems, components of toolkit systems are small and functionally independent. A consequence of structuring systems with autonomous units is that these systems are extensible. Toolkit systems thus may evolve over time, acquiring increased functionality as new tools are added. At the early stages of evolution, the toolkit is typically small and the tools serve more as valuable supplements to general-purpose tools. As the toolkit is gradually expanded, it will play a more dominant role in debugging and eventually general-purpose tools will become the last resort. For obvious reasons, however, it is inconceivable that toolkit systems will replace general-purpose systems. This shortcoming can actually be taken advantage of by the tool designers and implementors. Since the goal is to complement general-purpose tools rather than to eliminate them, development of toolkits may proceed incrementally, with attention focused on tools that give the greatest potential benefit. In addition, flaws found in early development efforts can be subsequently avoided, giving rise to higher quality tools.

In summary, the essence of toolkit systems is that each constituent tool incorporates an effective procedure for detecting a class of programming error. Hence, suspicion about the presence or absence of a particular error can be confirmed easily. From the perspective of the users, the designers, and the implementors, toolkit systems in many respects offer advanced features surpassing available alternatives. It is anticipated that

debugging time will be reduced drastically with a reasonably developed toolkit. However, toolkit systems should be regarded as complements to general-purpose systems, rather than as a complete solution exhausting all debugging needs. A detailed evaluation of the toolkit approach to debugging system design is presented in Chapter 4.

2.2. Human Engineering in Design

As mentioned earlier, consideration of human factors is a prerequisite to successful software tool development. With respect to debugging tools, this argument can be taken further: as manual methods are always viable and preferred alternatives, complicated tools will be abandoned regardless of sophistication in capability. Designing practical debugging tools must take user psychology into consideration. The first concern is to gain user confidence in using the tools.

Establishing user confidence means (among other things) that the tools can be trusted to perform designated tasks. In addition to being adequately robust and reliable, it is essential for the tools to tackle the problem *as a whole*. A common fault in tool design is that only special instances of the problem are handled. The solutions adopted are motivated primarily by premature concern for efficiency and secondarily by ease of implementation. It is sometimes forgotten that performance characteristics are meaningless unless requirements are first fulfilled. Our answer to attaining economy while insisting that tools meet their functional expectations is to combine reasonably efficient but simple algorithms with exhaustive methods, which are used only for a few difficult cases. Although tools which employ such a mixed strategy may not be the most efficient, the implementors will be able to deliver the expected capabilities without spending undue effort in finding the best algorithm.

Another important human concern is to minimize the effort needed to use the tools. The criterion is to substantially reduce, if not totally eliminate, human interaction during the course of exercising the tools. It has been explained earlier that special-purpose tools typically need minimal human guidance because of the built-in error detecting procedures. In fact, some special-purpose tools may carry out error analysis tasks without requiring explicit information from the user; the information needed is derived from other sources. It may seem that such sources are scarce, but as we shall see in Section 2.3, there are many ways to extract useful information from the program.

A final important human aspect is usefulness. In addition to the preceding ways to initiate users, tools must produce results useful for further debugging efforts in order to receive final acceptance. For error analysis tools, it means that the diagnostics produced can be easily related to the actual cause. Ideally, errors detected should be described in pragmatic terms, that is, in terms of how language features are misused in a particular context. Pragmatic error descriptions are more helpful for correction than symptomatic descriptions, which merely give an account of the external (observable) effects of the program at termination. The erroneous program fragment in Figure 2-1, which is intended to sort a vector by repeatedly moving the minimum element from a successively smaller vector slice to the head of the slice under consideration, serves to illustrate the difference between symptomatic and pragmatic error descriptions. An error is revealed upon examination of the vector processed by procedure SORT. (We encourage the reader to diagnose the error at this point.) The symptomatic error description might be "incorrect result," but the pragmatic cause is "missing declaration for variable I in procedure MIN." Correction is straightforward given the latter error description.

```
procedure SORT(var V : VECTOR);
var I : INDEX;

   function MIN(START : INDEX) : INDEX;
   var POSITION : INDEX;
   begin
     POSITION := START;
     for I:=START+1 to UPPER_BOUND do
       if V[I] < V[POSITION] then
         POSITION := I;
     MIN := POSITION
   end;

begin
  for I:=LOWER_BOUND to UPPER_BOUND-1 do
    SWAP(I, MIN(I))
end;
```

Figure 2-1: Erroneous Sort Procedure

Although pragmatic error descriptions are preferable from a debugging standpoint, it is in general impossible to derive such descriptions *precisely* from the source code alone without additional information or constraints on the various language entities. In the preceding example, it will be impossible for a mechanical tool to discover that the inner loop control variable must be a local entity, if such a restriction is relaxed by the language definition. Disregarding whatever information that can be obtained, the problem of determining precise descriptions of detected errors is inherently a very difficult one. As it occurs in the context of compilation, the problem has received considerable attention [53, 57, 64], but much remains to be learned[3]. Compared with syntax errors, the handling of programming errors is much more difficult. With this difficulty in mind, a more modest goal is to furnish diagnostics that are helpful, though perhaps imprecise.

[3]As a point of interest, some recent commercial compilers circumvent the problem by invoking a visual editor at the point of error and thus relying on the user to make the necessary corrections, which are typically straightforward. This solution might well be the preferred one in an integrated programming environment.

2.3. Taxonomy of Features

Toolkit systems may gradually evolve. It is important to delineate their functionalities not only to help visualize the limitations, but also to assist tool designers in selecting applicable features. An obvious way to define the boundaries of toolkit systems is to classify errors according to the nature of their causes. For example, an error in an expression may be traced to the operator used, giving the error type "Incorrect Operator Used", which can in turn be classified as "Computational Error". Such attempts have been reported by Boehm, *et al.* [10], Endres [23], Rubey [58], and Thayer, *et al.* [66]. "The principal problem encountered in such an approach is the tendency to create a category for each error analyzed." [66] Any earnest attempt results in a prohibitively large number of error types. The largest classification known lists over 400 error types [66]. It is doubtful whether such long lists are useful at all for our purpose. Another problem is that such classifications do not yield insights as to how each error type can be handled. The error type "Incorrect Operator Used" hardly suggests any useful interpretation for automatic detection. This approach is not adopted here.

The features discussed here are grouped into four classes: semantic error detection, consistency checking, code auditing, and data structure analysis; each is treated in a separate section. Other features are possible and may even be desirable. A more detailed classification will require practical experience with toolkit systems; it will not be attempted at this point.

A characteristic of the above classification is that each class represents the enforcement of some predefined rules. Thereby, means for detecting violations become apparent. In addition, the requirement to minimize user effort is easily satisfied. For

example, semantic error detection tools expose logic errors that manifest themselves by violation of the language semantics. Necessary information for their detection can be derived from the language definition without user participation.

2.3.1. Semantic Error Detection

A programming language specification must define rules governing syntax and semantics; any deviation from the prescribed rules constitutes an error. Syntax errors are much less relevant with respect to debugging and thus are not considered further. Semantic errors can be classified as either static or dynamic, depending on whether detection is possible at compile-time or at run-time. Compilers must perform complete static semantic analysis in order that code generation can proceed correctly. Dynamic semantic error analysis, however, is sometimes neglected due to the substantial execution overhead incurred from the addition of run-time checks[8]. Compilers should not be relied upon for detecting all language stated violations.

To illustrate the inadequacy of compilers as debugging aids, consider again the erroneous sort procedure in Figure 2-1. Strictly speaking, the oversight of not declaring the for-loop control variable is a violation of the standard Pascal language definition [5], and the error should have been caught by a conforming compiler. Unfortunately, our local production compiler accepted the illegal program without complaint. As another example, consider the following program fragment.

---

[8]Dynamic semantic errors that cannot possibly be checked for are indications of language design deficiencies.

```
procedure OUTER;
var I : INTEGER;

  procedure INNER(var V : INTEGER);
  begin
    V := 1
  end;

begin
  for I:=1 to 100 do
    INNER(I)
end;
```

The language definition clearly states that the **for**-loop control variable must not be subjected to explicit assignment and must not be passed as a reference parameter to a subprogram. Thus, this error can be detected rather easily and inexpensively during compilation. However, a student compiler, which claims close conformance to the standard and is well-known for its diagnostic capability, failed to recognize the error. Indeed with existing compilers, compliance to the standard appears to be the exception rather than the norm[5]. Consequently, tools for detecting semantic violations will form an important set of debugging aids.

Some dynamic semantic errors are listed in Table 2-1. They are grouped into two categories: control structure errors and type constraint errors (on data objects). Since a data type is characterized by a set of values and a set of operations, type constraint errors can be further divided into value constraint errors and operation constraint errors. Note that in designing tools for detecting dynamic semantic errors, error types should be selected such that they are known to occur frequently (based on experience or experimental data) and that they do not overlap with the capabilities

---

[5]Fortunately this unorthodox situation is gradually improving as recent language design efforts tend to be more rigorous, complete, and unambiguous in order to promote faithful implementation (cf. Ada [55]).

No corresponding case statement alternative for the selector value.
Function not returning a valid value.
A process attempting to communicate with another terminated process.

(a) Control Structure Errors

Assigning a value not in the valid range of the variable.
Index selectors not in the valid range.
Selecting fields from an inactive variant of a discriminated union.
Attempting to reference a variable/pointer that has an invalid value.
Division by zero.
Arithmetic overflow.
Arithmetic underflow.

(b) Type Constraint Errors

Table 2-1: Dynamic Semantic Errors

already offered by existing facilities (including but not limited to compilers). For example, arithmetic overflow errors are almost always trapped by the hardware whereas referencing uninitialized variables is often left to the programmer's discretion; a tool for detecting the latter error type is therefore much more useful.

2.3.2. Consistency Checking

Violating semantic rules is but one of many ways that logic errors may manifest themselves. Another perhaps more common way is for logic errors to exhibit themselves as inconsistencies. Inconsistent usage patterns in programs are frequently referred to as (program) anomalies. Anomalies may be symptomatic of potential problems, but are not necessarily errors in the strict sense. A trivial example of an anomaly is the absence of references to a declared variable. This might indicate a typographical error due to similar variable names, as the variables in the declaration

```
var UK_POPULATION, US_POPULATION : POPULATION;
```

A simple tool for checking declarations against references will detect and warn about the inconsistency. Should it happen to be an error, much of the debugging effort

A variable is assigned but never referenced subsequently.
No reference to a variable between two assignments.
Presence of unreachable statements.
Some fields of an active variant of a discriminated union are not assigned to before another variant is selected.
A recursive function that produces no side effect is invoked with the same values of arguments in successive calls.

(a) Program Anomalies

Interface consistency across separately compiled modules.
Data and control flow consistency between program and design.
Adherence to the pre- and post-conditions identified in the design.

(b) Checks based on Design

Table 2-2: Consistency Checks

normally required will be avoided. As another example, an unreachable-statement anomaly is present in

```
if (DAY >= 1) or (DAY <= 366) then
  SCHEDULE(DAY)
else
  WRITELN('Not in valid range.');
```

This is inconsistent with the purpose of an **if**-statement, and an examination of the **if**-expression will reveal the error. Again, in this case, much labor will be saved. Table 2-2(a) lists more program anomalies.

The extent of anomaly checks depends somewhat on the amount of redundant information made available by language designers. Type declaration is an example of redundancy, which allows illegal manipulations of logically incompatible objects to be caught. Although there is a growing trend to incorporate more secure features into programming languages, anomaly checks represent only part of all possible consistency checks that can be taken. Another major source of information that can be utilized for consistency checking is provided by the design documents. DACC [10], for

example, is a tool that demonstrates the feasibility and advantages of such an undertaking. Some consistency checks that can be performed based on the design documents are presented in Table 2-2(b). Lichtman [46] describes a methodology for detecting inconsistencies between a program and its design.

2.3.3. Code Auditing

Code auditing is the process of examining programs for coding malpractice according to a prescribed set of rules denoting proper usage, which is known as a programming standard. The purpose of such subjective, but generally well-accepted, standards is to assure uniform style and appearance of programs in order to promote reliability and maintainability. To these ends, items addressed by a standard include format of indentation, naming and documentation conventions, and usage restrictions on certain language features. Only the last of these items is of particular interest to debugging, though violations of the others might well impair debugging productivity.

Certain language features are often considered as harmful, yet programming convenience is sacrificed without them. **Goto** statements, side effects, aliasing, pointers, and global variables are a handful of well-known examples. Their harmful aspects are documented in detail in [22, 36, 54, 56]. Since these features are important sources of errors, violations of usage restrictions set by a programming standard serve as an early indication of problems, and should be taken as errors. For example, the expression

```
SEED + RANDOM(SEED)
```

may yield different values on different compilers, if function RANDOM modifies variable SEED. Debugging is necessitated when programs of this sort are transported

Default initialization value of variables.
Evaluation order dependencies.
Storage allocation and alignment for predefined types.
Execution timing of program constructs.
Strategy in choosing an alternative in constructs involving nondeterministic selection.

Table 2-3: Implementation-Dependent Features

to "incompatible" environments. In general, reliance on implementation-dependent[6] features is a dangerous practice. Table 2-3 lists some implementation-dependent features. In other cases, violations of the standard might well point out actual errors. For example, an infinite loop construct without exit violates any intent of finite computation, as in

```
while TRUE do
  WRITELN('Hello!');
```

The utility of code auditors as debugging aids is readily evident.

2.3.4. Data Structure Analysis

Thus far, attention has been focused on logic errors originating from the misuse of language constructs without considering the purpose for which they are used. A large number of errors occur in the context of implementing (abstract) data structures. Each data structure can be described by a set of structural properties and a set of primitive operations provided explicitly to the client for manipulation. This description serves as a basis for determining integrity, whether manually or otherwise. For a particular data representation, tool designers may develop a description that will facilitate the construction of tools for detecting integrity violations. Consider the case

---

[6]This term is used here to represent those implementation-defined attributes that are not warrant to be taken advantage of by programs. MAXINT, denoting the largest representable integer, is an example of an implementation-defined, but not implementation-dependent, attribute.

of a linear list implemented using pointers. Conceptually, a linear list *l* consists of an integer *count* of nodes and an ordered set of nodes satisfying the structural properties that

1. *count* is always nonnegative,
2. *l* is **nil** if *count* = 0,
3. *l* points to the first node if *count* > 0,
4. each node has a *link* field from which the successor node can be reached, and
5. the *link* field of the last node is **nil**.

The primitive operations of concern are

1. *insert(l, p)*, which increments *count* associated with list *l* by one,
2. *delete(l, p)*, which decrements *count* associated with list *l* by one,
3. *search(l, key)*, which returns in *q* the value **nil**, or the value of *l*, or the *link* field of some node of list *l*, and
4. *successor(l, p)*, which returns in *q* either the value **nil** or the *link* field of some node *p* of list *l*.

The above description can be made more elaborate or general depending on the level of checks desired; it suffices here to illustrate the essence of such descriptions. Tools can now be constructed to check for structural conformity with respect to the above description. These tools will be able to detect improper manipulation of the underlying data representation initiated by both the implementor of the data structure and the client of the data structure if direct access to the representation is possible.

As an example to illustrate the utility of data structure analysis tools, suppose that in implementing the operation INSERT_TO_HEAD for linear lists, the LINK of the newly added node is not updated to connect with the rest of the list. This oversight

can be detected by a tool which compares the actual number of nodes with the *count* of the list. As another example, suppose that when restructuring the links in the operation DELETE_NODE, the correct statement

PREDECESSOR↑.LINK := CURRENT↑.LINK

is inadvertently written as

CURRENT↑.LINK := PREDECESSOR↑.LINK

A circular list has now resulted. This mistake can be detected by the same tool used previously. The trick lies in the implementation of the tool itself. Instead of comparing the count at the end of a full traversal of the list, the tool will stop and report an error as soon as *count* is exceeded. A more sophisticated version will be able to report the circular condition by maintaining a list of distinct pointers encountered in the traversal. This points out that through careful design a single tool can be used to detect different pragmatic errors.

In order to use such tools, the correspondence between the conceptual entities in the descriptions and the actual entities used in the implementation must be given. The exact nature of specifying the correspondence is left to the tool designers. Note that the correspondence need not be one-to-one. For instance, the conceptual entity *count* above may not have a counterpart in an implementation, in which case the tool will create and maintain such an item for internal use.

With a small, fixed tool set, debugging support will likely be limited to some primitive structures, such as lists, trees, stacks, and queues. It is hoped that the continuing research on data structure specification techniques will bring more insights to the problem.

2.4. Program Analysis Methods

The features offered by a toolkit system are the types of errors that the system is able to detect. In implementing a tool, different program analysis methods for detecting errors may be adopted. In order to build useful tools, a thorough understanding of the tradeoffs between alternative program analysis methods is necessary. Program analysis methods are divided into two broad categories: static and dynamic; each is examined in turn.

Static analysis methods can determine the presence or absence of certain types of errors in a program without actually executing the program. Many mistakes, such as omitting references to declared variables and data type mismatches between formal and actual parameters, can be detected easily and dependably by a textual scan on the program. More involved static error analysis can be performed using program flow analysis techniques. Flow analysis techniques can discover errors that can be represented as a sequence of events. For example, the event sequence {undefine, reference} on variables denotes accessing undefined values. This type of error can be detected by flow analysis, with some limitations as noted below. See Hecht [34] and Muchnick and Jones [50] for a more thorough treatment on the subject. Essentially, static analysis methods work on a formal program model, such as parse tree or graph representation, and involve a simulated execution of the program over all program paths. Therefore, the result of analysis is applicable to all possible program executions irrespective of the input data. It follows that if the presence of certain errors is not shown, their absence can be assumed. Furthermore, algorithms developed for static analysis are generally efficient, making them attractive from a computational standpoint. However, static analysis methods suffer from many theoretical limitations. Many properties of a program cannot be determined in general

by merely examining the program text. For example, complete variable aliasing information and feasibility of program paths cannot be known statically. These problems are handled by making worse case assumptions, and, as a result, insignificant diagnostic messages will be generated. Experience has shown that existing static analysis tools tend to produce a massive amount of superfluous messages, which will distract, or even annoy, the user. This is the main drawback of static methods.

Dynamic analysis methods can discover the presence of errors through execution of the program under analysis. The detection of errors in dynamic analysis is accomplished by adding monitoring code (run-time checks) to the program. The resulting program is then executed. By executing the program, many properties of a program unknown by static methods can now be determined. Diagnostics will only be generated for actual errors encountered rather than for all possible errors as generated by static methods. The reduction of superfluous diagnostics is the major advantage over static methods. Dynamically produced diagnostics, however, are only relevant to the particular execution path caused by the given input data. Hence, dynamic analysis methods can show the presence of errors with respect to the given data sets, but cannot guarantee the absence of errors. Demand for resources is also relatively higher for dynamic methods, although this should only be of secondary importance.

A debugging tool for a given error type may be best implemented using either static methods or dynamic methods or a combination thereof. If an error type is amenable to positive identification by static methods, then only static methods are considered. In this case, all errors detected will be reported even though they might be irrelevant to the particular execution under consideration. The reason for not supporting execution error analysis is to encourage the user to remove all known errors before

proceeding further. This restriction on how tools can be used is in support of the view that the early identification of errors will lead to improved software reliability, and with obvious cost advantages. In spirit, it is similar to strong typing. When static means alone are inadequate, an integrated static and dynamic method can be used. Static analysis is first applied to achieve a thorough program analysis. Errors found at this stage can be classified as either *definite* or *potential*. A statement in a program is said to produce a definite (respectively, potential) error if its execution will always (respectively, sometimes) cause the same error to occur. All definite errors found should be reported because it is reasonable to assume that each statement in a program lies on some executable path. Potential errors, on the other hand, should be monitored with run-time checks as it is unclear when these errors will be triggered. Run-time checks for definite errors are also inserted to test if they will cause run-time errors for some particular execution. If any run-time checks are inserted, the modified program will then be executed (the dynamic analysis phase) to obtain a profile of execution errors (if any). In order to avoid overwhelming the user with diagnostics, only definite errors and the first execution error will be reported. Of course, the reporting of potential errors and subsequent execution errors may be optionally selected by the user. Such a selection may be done via a tool option, say DIAGNOSTICS. The possible values for this option are *Terse* and *Full*, whose meanings are given below.

Terse	provide short diagnostics. (Report only definite errors and first execution error.)
Full	provide comprehensive diagnostics. (Report all definite, potential and execution errors.)

Although this option is only meaningful for tools employing both static and dynamic analysis, extending it to the entire collection of tools will give a more unified view of the system. For tools that employ only static analysis, this option has no effect other than to maintain interface consistency.

When an integrated static and dynamic method is used to implement a tool, there are other selection possibilities for the diagnostic messages. The selection for the diagnostics from static analysis can be *Terse*, *Full*, or *Ignored*, with the following meaning:

Terse	report only definite errors.
Full	report definite and potential errors.
Ignored	do not report any definite or potential error.

These three choices apply also to diagnostics from dynamic analysis, with the following meaning:

Terse	report first execution error.
Full	report all execution errors.
Ignored	do not report any execution error.

By using two tool options (one for diagnostics from static analysis and the other for diagnostics from dynamic analysis), a greater range of possibilities for diagnostic reporting can be provided. Although using an extra tool option will give the user more flexibility, it is not apparent in this case whether the added flexibility will be of any practical value. In fact, one might argue that it will be a mere source of confusion for someone who is unfamiliar with notions of static and dynamic analysis. Therefore, the original choice of using a single tool option should be retained. When the benefits of adding more features to a tool (by means of tool options) are unclear, a simpler interface should be opted for. Adding tool options freely is indeed a bad practice, as Kernighan and Pike [45] commented on some UNIX[7] tools: "Creeping featurism encrusts commands [tools] with options that obscure the original intention of the programs."

[7]UNIX is a registered trademark of Bell Laboratories.

2.5. Implementation

2.5.1. Model

A toolkit system, as viewed by the user, is a black box which takes as input a program and produces as output an error report. (The report is specific to a particular type of error, depending on the tool selected.) This view is depicted in Figure 2-2(a). Internally, static and dynamic analysis are performed on the given program. Static analysis reports program errors that are detected by examining the source text, and inserts run-time checks to monitor errors in execution. Dynamic analysis reports errors that are caught by the inserted run-time checks during the execution of the modified program. Prior to the actual static analysis, parsing is first done to convert the flat source text into a more convenient form for manipulation. Tools for these tasks are the parser, the static analyzer, and the code executor. Their organization is shown in Figure 2-2(b).

For each error type that the system is able to detect, a separate static analyzer is implemented. It is very important to standardize the input and output representations used by the collection of static analyzers, although it is tempting for convenience reasons that different types of analysis employ their own esoteric representations. The use of standardized representations allows all static analyzers to share the same parser and code executor, thus reducing the implementation effort. More importantly, it provides a common basis for independent implementors to communicate and understand the work of others. This point is particularly important to the long term viability of the system as maintainability becomes the crucial factor.

We choose the parse tree (abstract syntax tree) as the unique internal representation. There are several reasons for this choice. As the input representation to static

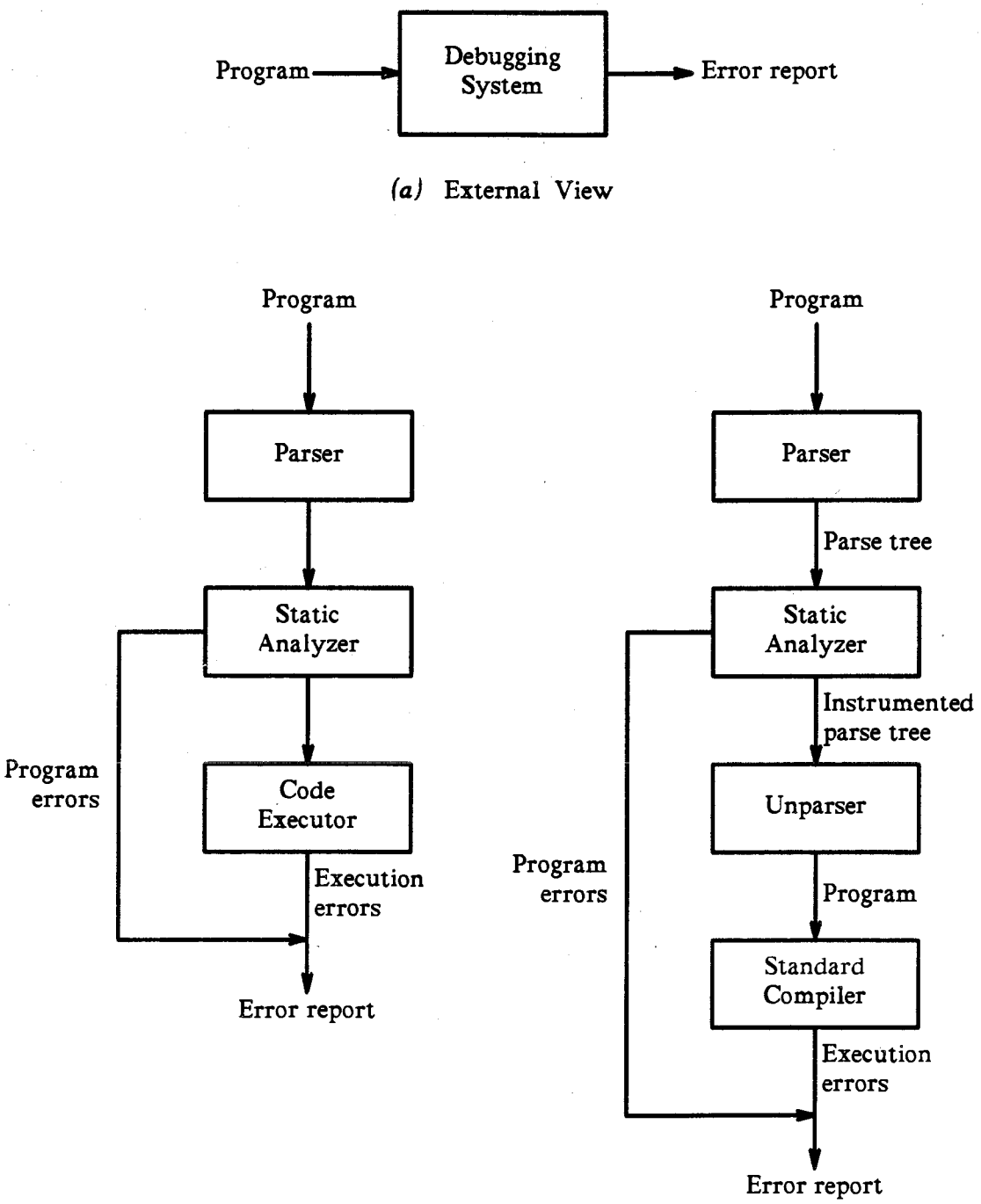

Figure 2-2: Structure of Debugging System

analyzers, a parse tree captures all information of the original program and is a useful structure for performing general analysis. The work of Cameron and Ito [11] has demonstrated that the parse tree is a convenient basis for performing static analysis (and program manipulation in general). More specialized representations, such as the call graph and flow graph combination that is used widely in flow analysis applications, are less desirable because it is unclear whether they are suitable for general analysis. As the output representation of static analyzers, a parse tree offers a simpler alternative for implementing dynamic analysis. Instead of writing an intricate code executor, an existing compiler can be taken advantage of, with a straightforward unparser (prettyprinter) serving as the interface. The program resulting from unparsing can also be made accessible to the user, allowing him the flexibility of retaining the run-time checks as a permanent part of his program (for testing purposes, for instance). Should efficiency become a concern, he may delete part of the checks. This possibility is feasible because the user is dealing with a program at the source level, not some obscure intermediate language. Finally, the use of a parse tree as the unique internal representation means that static analyzers can be directly cascaded together without additional processing. Figure 2-2(c) illustrates the proposed implementation model. Note that this model represents only the kernel of the system; additional elements, including a user interface, are needed to create a stand-alone and a more usable system.

2.5.2. Program Instrumentation Techniques

Program instrumentation refers to the insertion of source statements into a program for information gathering purposes [41]. It is commonly used as a means to collect execution statistics about a program (*program profiling* as it is known) [28, 30]; our use of it is for detecting execution errors. We now discuss some useful ideas about program instrumentation in this narrow sense.

An error occurs in a program's execution when the state of computation violates certain expected (predetermined) conditions. These conditions can be expressed in terms of certain program entities. Errors in execution are detected by monitoring these entities. To do this, we associate a *monitor variable* with each such entity. A monitor variable is used to capture the abstract state of the associated entity during execution. The actual monitoring is carried out by inserting statements into the program. For this reason, such inserted statements are called *monitor statements*. We shall use the more intuitive term run-time checks to mean monitor statements when there is no need to be more precise.

The value of a monitor variable denotes the abstract state of the associated entity. The possible states that a monitor variable may assume (that is, the characteristics of the associated entity that are of interest) are known in advance. For many practical purposes, the set of possible states is enumerable and is typically small in number. Some of the states may be designated as *error states*. When a monitor variable enters into an error state, a message to that effect is emitted. For example, to detect whether a function returns a value, two states, *FunctionAssigned* and *FunctionNotAssigned*, would suffice, with the latter being an error state. In Pascal, the set of possible states of a monitor variable can be denoted by an enumeration type. Identifiers for monitor variables should be selected such that they can be easily distinguished from the ordinary program variables. Using a unique name suffix will serve the purpose. For example,

```
type FUNCTION_RETURN_MONTYPE =
        (FUNCTION_ASSIGNED, FUNCTION_NOT_ASSIGNED);
var  FACTORIAL_MON : FUNCTION_RETURN_MONTYPE;
```

where FACTORIAL_MON is the monitor variable for the function FACTORIAL.

When monitor variables are being associated with program variables, care must be taken to ensure that the proper monitor variable associations are maintained throughout program execution. The identity of a monitor variable may be confused when the proper monitor variable to be used is not known statically, but must be determined at run-time. In Pascal, this happens with the monitor variables of array elements, dynamically allocated objects (called *heap objects* henceforth), and parameters that are passed by reference. Some possible solutions for disambiguating such monitor variables at run-time are presented below.

Case 1: monitor variables of array elements.

Suppose that each element of an array has an associated monitor variable and that we want to identify the monitor variable of an array element whose indices are given by expressions that cannot be evaluated statically. The solution is straightforward. Temporary variables are first established to hold the values of the indexing expressions. The indices of the array element in question are then replaced by the respective temporary variables. These temporary variables can now serve to identify the proper monitor variable. Alternatively, the indices can be recomputed, thus saving storage at the expense of efficiency. This solution is applicable when the indexing expressions are free of side effects.

Case 2: monitor variables of heap objects.

Heap objects are created and destroyed dynamically. Aliases to them can exist as a result of pointer assignments. A simple method that allows the monitor variables of a heap object to be identified easily is to package the heap object and its monitor variables into the same data aggregate by

modifying the declaration of the heap object. An example will clarify the point. Consider the following declarations.

```
type LIST = ↑NODE;
     NODE = record
              INFO : INTEGER;
              LINK : LIST
            end
```

Suppose a monitor variable is desired for INFO. This is done by changing the declaration of NODE to

```
NODE = record
         INFO : INTEGER;
         LINK : LIST;
         INFO_MON : SOME_MONTYPE
       end
```

With this method, the monitor variables of a heap object can be assessed as long as a pointer to that heap object is known. Even when pointer aliases are present, the proper monitor variable can always be identified.

Case 3: monitor variables of reference parameters.

If a variable under monitoring is being passed to a subprogram via call by reference, then monitoring must also be taken on that formal parameter of the called subprogram. Since a formal reference parameter is just an alias of the corresponding actual parameter and an operation on any alias will affect the same datum, it must be ensured that the monitor variables of the aliases will remain consistent at all times; that is, any change to the monitor variable of an alias must be propagated to all other monitor variables of the alias group. One solution is to ensure that all aliases share the same monitor variable. To do this, aliases among monitor variables can be created

using the same parameter passing mechanism. Specifically, the relevant subprogram calls are modified to include the monitor variable of the actual parameter as an extra argument that is also passed by reference. The corresponding subprogram heading must be modified accordingly. For example, consider the subprogram call

```
FOO(X)
```

where the heading of FOO is

```
procedure FOO(var I : INTEGER);
```

The necessary modifications are shown below.

```
FOO(X, X_MON)
procedure FOO(var I : INTEGER; var I_MON : SOME_MONTYPE);
```

Note that to maintain semantic validity, all calls to FOO must be modified even if the argument of some calls does not need to be monitored. In that case, a dummy monitor variable can be used.

We now turn our attention to monitor statements. There appear to be two useful types of monitor statements, namely, *state-assignment statements* and *error-report statements*. State-assignment statements are used to maintain monitor variables in the proper state. In particular, they serve as initializations to monitor variables. Error-report statements are used to report errors when erroneous conditions are encountered. Each of these two types of monitor statements can be further classified as either *conditional* or *unconditional*. An unconditional monitor statement causes the intended action to be performed whenever that monitor statement is executed. In contrast, the action of a conditional monitor statement will be executed only if certain conditions are first satisfied. The choice of the types of monitor statements and the appropriate

places for their insertion are determined by static analysis. Usually, there is more than one possible solution.

As an example, we shall continue with the "function returning value" problem. Consider the following function for computing the factorial of a number.

```
function FACTORIAL(I : INTEGER) : INTEGER;
begin
  if I <= 0 then
    WRITELN('Input must be greater than 0')
  else if I = 1 then
    FACTORIAL := 1
  else
    FACTORIAL := FACTORIAL(I-1) * I;
end;
```

A smart static analyzer would produce

```
function FACTORIAL(I : INTEGER) : INTEGER;
begin
  if I <= 0 then
    begin
      WRITELN('Input must be greater than 0');
      { unconditional error-report statement }
      WRITELN(ERROR_FILE,'FACTORIAL not returning value.')
    end
  else if I = 1 then
    FACTORIAL := 1
  else
    FACTORIAL := FACTORIAL(I-1) * I;
end;
```

while a straightforward one might arrive at

```
function FACTORIAL(I : INTEGER) : INTEGER;
var FACTORIAL_MON : FUNCTION_RETURN_MONTYPE;
begin
  { unconditional state-assignment statement }
  FACTORIAL_MON := FUNCTION_NOT_ASSIGNED;
  if I <= 0 then
    WRITELN('Input must be greater than 0');
  else if I = 1 then
    begin
      FACTORIAL := 1;
      FACTORIAL_MON := FUNCTION_ASSIGNED
    end
  else
    begin
      FACTORIAL := FACTORIAL(I-1) * I;
      FACTORIAL_MON := FUNCTION_ASSIGNED
    end;
  { conditional error-report statement }
  if FACTORIAL_MON = FUNCTION_NOT_ASSIGNED then
    WRITELN(ERROR_FILE,'FACTORIAL not returning value.')
end;
```

Note that the monitor statements used here are simple statements. In more complex situations, a monitor statement might involve a series of computations. Using the subprogram facilities to convey the monitoring action would be more appropriate in such cases.

We have attempted to shed some light on the instrumentation process. This is only the beginning of the investigation. A more formal and complete characterization is beyond the scope of this thesis.

2.5.3. Efficiency Considerations

There is usually more than one way to instrument a program for detecting a certain error. The overall cost of a method can be characterized by the amount of time spent in analysis and the execution overhead incurred from running the instrumented program. Analysis time and execution overhead are inversely related; a sophisticated

analysis takes more time than a straightforward one, but it would reduce the need for monitor statements which would in turn contribute to a decrease in execution overhead, and vice versa. The instrumentation example of the factorial program in the preceding section illustrates this point. The first instrumented program uses one monitor statement compared to four in the other one, but achieving the reduction requires a more costly analysis. Tool designers must consider both cost factors in selecting an economical design. Ideally, of course, the investment in analysis should never exceed the expected gain in execution cost.

A more involved analysis will likely reduce the number of monitor statements needed, but the corresponding decrease in execution time is not necessarily proportional. In fact, such occurrences should be regarded as coincidence. Empirical evidence has shown that as much as 90% of execution is spent in 10% of code. Although these figures are not conclusive, it does point out that reducing monitor statements has only marginal value unless it takes place in frequently executed regions of the program.

Despite that the number of monitor statements is a deceiving indicator of execution overhead, it is also the most accessible quantitative measure available to tool designers. The rule to bear in mind is never overwork the analysis for the mere sake of reducing monitor statements. When the benefits of reducing monitor statements are unclear, other indirect factors, such as the implementation complexity of a sophisticated analysis algorithm, can be taken into consideration in the cost tradeoff process.

2.6. Pragmatic Issues

A toolkit system is intended for interactive debugging. When an error is discovered, the user would examine the program output, the source code, and/or related documentation to gather hints about the error. An error hypothesis would then be devised, perhaps with the aid of other tools in the programming environment. A toolkit system can now be used to verify the hypothesis, if the system is equipped with such a capability. A possible scenario of using a toolkit system is as follows. Through a menu-driven interface, the user selects the desired tool. After the selection is made, the system might present some simple queries, such as

- What is the desired level of diagnostic report?
- What is the name of the erroneous program?
- What are the input data files for the program?

Of course, the answers to these queries can be stored and changed only as required, saving the user from repeatedly giving the same responses in a debugging session. When the necessary information is gathered, analysis of the given program is performed. The result of the analysis is turned into a report for presentation. The user then acts upon the report as appropriate, and the debugging cycle can now be repeated.

Besides interactive debugging, a toolkit system can be utilized for other purposes. We consider two possibilities. The first possibility is to supplement existing compilers with additional error detection capabilities. The dynamic semantic error detection tools will be of particular use here. In a loose programming environment, selected toolkit components can be grouped together via some specially created "shell". It can then be treated as a preprocessor to the compilers. Another more ideal situation is

that after parsing and static semantic analysis, the compiler would directly access the desired static analyzers to insert run-time checks before proceeding further with the normal compilation procedure.

Another possibility of use is for program testing. Traditional black box testing can be facilitated by inserting run-time checks into the program before testing begins. After the program has been exercised with some sample test data, the user will first turn to the error report. If errors are found, corrective action can be taken immediately. The usual step of examining the program input and output for correctness is thus by-passed. Alternatively, the toolkit components can be adapted to report errors interactively. The acknowledgement of errors at an earlier stage will save both time and effort in testing.

CHAPTER 3

EXAMPLE TOOLKIT COMPONENTS

I expect programming languages to do their utmost to protect me from myself - from the many stupid errors I am bound to commit when I program.

Jonathan Amsterdam [4]

This chapter presents the design of three tools: uninitialized variable access detection, bounded execution failure diagnosis, and parameter usage checking. These tools have been implemented as stand-alone instrumentation utilities using the program manipulation facilities provided by a Pascal metaprogramming system [12].

3.1. Uninitialized Variable Access Detection

Uninitialized variable access refers to the use of a data object's value before a valid value has been assigned. It is a common programming mistake as well as a semantic violation in virtually all programming languages. Unfortunately, few compilers are equipped to handle the error. It is even more discouraging that several language standardization efforts, including Ada [55] and Pascal [5], have decided to allow conforming compilers to ignore the error. Designing a tool to detect uninitialized variable accesses is a worthwhile undertaking since such a capability is not likely to be found in compilers, not even validated ones.

An obvious method for detecting uninitialized variable access is to use exhaustive monitoring, which involves inserting statements into the program to monitor all variable accesses and variable assignments. One disadvantage of this strategy is that

because the relationships among variable assignments and accesses are not computed, it is not possible to provide the user with an intelligent static diagnostic report which warns about errors and probable errors; the discovery of uninitialized variable accesses will depend entirely on dynamic analysis. Another disadvantage is that the execution overhead incurred from exhaustive monitoring may be unacceptably expensive in some computation intensive applications. When a static diagnostic report is desired or when execution overhead is a concern, the use of a more sophisticated analysis algorithm is warranted. The objectives in designing such an algorithm are to provide a concise static diagnostic report and to reduce the amount of run-time checks inserted into the program.

Data flow analysis is applicable to the detection of uninitialized variable access, although traditionally it is used for program optimization. The relevant data-flow problem is *use-definition chaining* (ud-chaining). The ud-chaining problem is to compute, for each use of a variable in a program, the list of *definitions* that can *reach* that use. This list of definitions for a use is called the ud-chain for that use. A definition is a statement which attributes a value to a variable, such as an assignment or a **read** statement. A definition is said to *reach a use* if the use may potentially refer to the value attributed by the definition. To compute the ud-chains, the standard data-flow algorithm, *reaching-definitions*, can be used. The reaching-definitions algorithm is described in detail in Aho and Ullman [3] and Hecht [34]. The ud-chaining problem can be applied to detecting uninitialized variable access as follows. Before computing the ud-chains, introduce dummy definitions for all variables at the beginning of the program. After the ud-chains are computed for the modified program, they can be examined for error conditions. If a ud-chain contains a dummy definition, then there is a possibility that the use corresponding to that ud-chain is erroneous. (Note that a possibility for error does not mean an error will

necessarily occur at run-time. In particular, the use will never be in error if there is no executable path between the dummy definition and the use.)

Recall from Section 2.4 that a statement in a program is said to produce a definite (respectively, potential) error if its execution will always (respectively, sometimes) cause the same error to occur. The ud-chains are useful in determining the statements (uses of variable) that will produce definite or potential uninitialized-variable-access errors. In particular, if a ud-chain contains only dummy definitions, then the corresponding use will produce a definite error. Similarly, a use will produce a potential error if its ud-chain contains a dummy definition. Using the ud-chains in this manner to generate static diagnostics, however, could lead to redundant warnings. For example, if the variable X in

```
QUOTIENT := X div Y;
if QUOTIENT >= 0 then
  MODULO := X - QUOTIENT * Y
else
  MODULO := X - (QUOTIENT - 1) * Y
```

is not initialized, then each of the three uses of X above would be reported as a definite error, but reporting the first occurrence is sufficient to warn the user about the error. In addition, the ud-chains do not give direct information as to where monitor statements can be eliminated. While it is possible to design an algorithm which uses the ud-chains to reduce redundant diagnostics and run-time checks[8], we suspect that the resulting solution for the problem at hand will be neither simple nor cheap. These factors prompt the design of a new algorithm in favor of modifying the ud-chaining solution to fit our needs.

---

[8]One solution that comes to mind is to use the topological ordering of the flow graph together with the ud-chains to decide where warnings and/or monitor statements are needed or not needed. When procedure calls are present, interprocedural analysis is required; it is not entirely clear how to handle side effects, aliasing, and recursion with this method.

Some features of Pascal, which are also typical of other common programming languages, hinder the design of an algorithm for statically detecting uninitialized variable access. The most difficult-to-handle features are heap objects and arrays. For simple, semistatic variables[9], such as variables of INTEGER type, there is no ambiguity as to which data object is being referred to by a variable access. This is not the case with variable access to heap objects or array elements because the same textual representation of a variable access, such as A[I], can and will denote different data objects at run-time. Static methods are incapable of resolving this identification ambiguity, let alone determining the validity of such variable accesses.

The scheme adopted here involves the use of exhaustive monitoring and a supplementary algorithm, which we shall refer as the *Reduction Algorithm*. Exhaustive monitoring will be used exclusively when a static diagnostic report is not desired and reducing execution overhead is not a concern. Otherwise, exhaustive monitoring and the Reduction Algorithm will be used in a complementary fashion, where the former is responsible for handling the difficult language problems and the latter takes care of the remaining language features. Specifically, exhaustive monitoring is used to handle structured variables and programs with arbitrary gotos, while the Reduction Algorithm is used to handle unstructured variables (including pointer and set variables) in programs which use only certain restricted forms of gotos or no gotos at all. Unlike exhaustive monitoring, the Reduction Algorithm computes variable usage information which can be used to generate static diagnostics and to guide the insertion of monitor statements. Details of the algorithms are given in Appendix A.

---

[9]Semistatic variables refer to the class of variables whose lifetime, size, and relative location in the activation record are known at compile-time. In Pascal, all variables except heap objects are semistatic.

We now examine the operation of a program instrumentation tool which employs both exhaustive monitoring and the Reduction Algorithm. When the tool is invoked, the following queries will be made to the user:

- Would you like a static diagnostic report on unstructured variables?

  If the answer is positive, then the Reduction Algorithm will be used and static diagnostics will be produced. If the answer is negative, then no static diagnostics will be produced but the Reduction Algorithm may still be used. See the last query below.

- Would you like the diagnostics to be reported in full or in terse form?

  This option affects the diagnostics (if any) from both static and dynamic analysis. See Section 2.4 and the example diagnostic reports below.

- Is it desirable that the execution overhead of the instrumented program be reduced?

  This query is presented only if the answer to the first query is negative. It determines whether the Reduction Algorithm should be used.

To show some flavor of the diagnostics produced, consider the following program:

```
     program EXAMPLE(INPUT,OUTPUT);
     type
       LIST = ↑NODE;
       NODE = record
                NUM : INTEGER;
                LINK : LIST
              end;
     var
       NUMBERS, ENTRY : LIST;
       FOUND, LAST : BOOLEAN;
       NEWNUM : INTEGER;

       procedure SEARCH(L : LIST; NUMBER : INTEGER;
                        var SUCCESS : BOOLEAN);
       var CURR : LIST;
       begin
1        SUCCESS := FALSE;
2        CURR := L;
```

```
3         while CURR <> nil do
4           if CURR↑.NUM = NUMBER then
              begin
5               SUCCESS := TRUE;
6               CURR := nil
              end
            else
7             CURR := CURR↑.LINK
        end;

      begin
1       NUMBERS := nil;
2       repeat
3         SEARCH(NUMBERS,NEWNUM,FOUND);
4         if FOUND then
5           LAST := FALSE
          else
            begin
6             NEW(ENTRY);
7             ENTRY↑.NUM := NEWNUM;
8             ENTRY↑.LINK := NUMBERS;
9             NUMBERS := ENTRY;
10            if LAST then
11              WRITELN('Consecutive new entries.')
              else
12              LAST := TRUE
            end
        until EOF
      end.
```

The program contains two uninitialized variables: NEWNUM and LAST.

Suppose the user wants a static diagnostic report.

1. When the user selects terse diagnostics, the static diagnostics produced are

```
Uninitialized Variable Accesses Detected:
  Variable 'NEWNUM' of subprogram 'EXAMPLE' -
  An execution error will result when the variable is first referenced in
    Subprogram 'EXAMPLE' statement 3 - used as a value parameter.
```

and the execution diagnostics produced are

Execution Errors Detected:
Accessing uninitialized variable at statement 3 of subprogram 'EXAMPLE'.

Program execution is halted.

2. When the user selects full diagnostics, the static diagnostics produced are

Uninitialized Variable Accesses Detected:
Variable 'NEWNUM' of subprogram 'EXAMPLE' -
An execution error will result when the variable is first referenced in
Subprogram 'EXAMPLE' statement 3 - used as a value parameter.

Warnings:
Variable 'LAST' of subprogram 'EXAMPLE' -
The following references to the variable may be invalid:
Subprogram 'EXAMPLE' statement 10.

and the execution diagnostics produced are

Execution Errors Detected:
Accessing uninitialized variable at statement 3 of subprogram 'EXAMPLE'.
Accessing uninitialized variable at statement 10 of subprogram 'EXAMPLE'.

Two points are worth noting about the static diagnostics produced by the Reduction Algorithm.

1. Although there are two references to NEWNUM (at statements 3 and 7), only the first error at statement 3 is reported. This is so because the algorithm has determined that the reference at statement 3 will always be executed before the one at statement 7, and hence, reporting the erroneous reference at statement 7 is redundant.

2. By examining the program carefully, we know that on the first iteration of the **repeat** loop, FOUND will be false and an erroneous reference will then be made to LAST. The Reduction Algorithm is incapable of discovering this fact, but it manages to report that the reference to LAST could be in error. We suspect that the same difficulty will be encountered by other data flow based algorithms because they are incapable of "understanding" the semantics of a program.

The instrumented program for the example is given in Appendix B.1. Note that instrumentation is performed in such a manner that when an uninitialized variable is

detected at run-time, its associated monitor variable will be set to a valid state so that further errors on the same variable will not be generated. For example, even though the erroneous reference to NEWNUM occurs within a loop, only one execution error is generated for it.

We conclude this section by presenting some statistics of the Reduction Algorithm in reducing monitor statements. The test data set consists of the following programs:

EightQueen	a recursive solution to the eight queens problem
HeapSort	heap sort of an array
Btree	B-tree search, insertion, and deletion routines
PureLisp	a Pure Lisp interpreter

The first three programs are taken from [69] and the Pure Lisp interpreter is a public domain program. These programs are free of uninitialized variable access errors. The statistics obtained are summarized in the following table:

	Exhaustive Monitoring Only		With Reduction Algorithm		Amount of Reduction with respect to	
	UV Ops	Total Ops	UV Ops	Total Ops	UV Ops	Total Ops
EightQueen	19	33	0	14	100%	58%
HeapSort	41	61	0	20	100%	67%
Btree	222	349	8	135	96%	61%
PureLisp	360	637	9	286	98%	55%

where

UV Ops = (Unstructured Variable Operations) the number of assignments and references to unstructured variables which require monitoring, and

Total Ops = (Total Operations) the number of assignments and references to all program variables which require monitoring.

3.2. Bounded Execution Failure Diagnosis

Execution of a program will fail when the program has exhausted all its allotted processor time or memory space. Such a condition is called a *bounded execution failure*. One reason for this failure may be that the program has been given insufficient resources (because the resource requirement of the program is unexpectedly high or the algorithm used is overly inefficient.) Another, perhaps more likely, reason is that the program contains some infinite computation. When a bounded execution failure occurs, the programmer may wish to know if there is indeed an infinite computation. This section presents a tool to facilitate the detection of the two most common sources of infinite computation, namely, infinite loops and infinite recursion.

The problem of determining whether or not a computation will terminate is well-known to be undecidable. Thus, it is in general impossible to mechanically detect the presence of infinite loops or infinite recursion. To circumvent this difficulty, a more restrictive notion of infinite computation is adopted. Specifically, a loop is considered infinite if the number of run-time repetitions exceeds a certain user-defined limit. Similarly, a subprogram invocation is considered as an infinite recursion when the number of outstanding calls (ie., the depth of recursive calls) on that subprogram exceeds the limit set by the user. Given these restrictions, detection of infinite loops and infinite recursion can be accomplished by monitoring, respectively, the number of loop iterations and the depth of recursion. We now describe the operation of a program instrumentation tool for this purpose.

When the tool is invoked, the user will be queried about the limits on loop repetitions and depth of recursive calls. After answers to these questions are

supplied, the tool will proceed to analyze and instrument the program. The analysis part involves associating a counter variable with each loop and each subprogram. Currently, only language-defined structured loops are taken into account; no provision is made to monitor loops that are simulated with **goto** statements. Instrumentation of the program is illustrated by the following transformations:

1. Monitoring of loop

```
while E do          ==>   LOOP_COUNTER_1 := 0;
  SOME_PROCESS            while E do
                          begin
                            if LOOP_COUNTER_1 = LOOP_LIMIT then
                              LOOP_ERROR
                            else
                              LOOP_COUNTER_1 := LOOP_COUNTER_1 + 1;
                            SOME_PROCESS
                          end
```

for-loops and **repeat**-loops are transformed similarly.

2. Monitoring of recursion

```
procedure P;              procedure P;
begin               ==>   begin
  SOME_PROCESS              if COUNTER_FOR_P = RECURSION_LIMIT then
end;                          RECURSION_ERROR
                          else
                            COUNTER_FOR_P := COUNTER_FOR_P + 1;
                          SOME_PROCESS;
                          COUNTER_FOR_P := COUNTER_FOR_P - 1
                        end;
```

(The actual instrumented code includes additional statements to keep track of statement locations for error reporting purposes. See Appendix B.2 for examples.) Note that initialization of a loop counter variable is performed every time its associated loop is entered, but subprogram counters are initialized only once in the main program.

The instrumented program produced can now be compiled and executed. If, at run-time, any counter variable attempts to exceed the preset limit, the running program will be terminated and an error report will be given. Several examples are given below to show the diagnosis produced: the actual instrumented programs for these examples are presented in Appendix B.2. In the examples, 10000 is used as the limit for both loops and recursion.

1. Infinite loop

```
        program EXAMPLE1(INPUT, OUTPUT);
        var I, J, K : INTEGER;
        begin
1         J := 0;
2         K := 0;
3         for I:=1 to 100 do
4           if I = 24 then
5             repeat
6               J := J + 1;
7               if J = 15 then
8                 while TRUE do
9                   K := K + 1
              until FALSE
        end.
```

Execution of the instrumented program produces the following error report:

```
The loop beginning at statement 8 in 'EXAMPLE1' has exceeded its
iteration limit of 10000.
Iteration status of outer loops:
        Loop at statement 3 has iterated 24 times.
        Loop at statement 5 has iterated 15 times.
```

2. Infinite recursion involving only one subprogram

```
program EXAMPLE2(INPUT, OUTPUT);

  procedure RECURSIVE;
  begin
    RECURSIVE
  end;

begin
  RECURSIVE
end.
```

The error report given is

```
Subprogram 'RECURSIVE' has exceeded its recursive call limit of 10000.
The subprogram is self-recursive.
```

3. Infinite recursion involving several subprograms

```
program EXAMPLE3(INPUT, OUTPUT);

  procedure MUTUAL2; FORWARD;
  procedure MUTUAL3; FORWARD;

  procedure MUTUAL1;
  begin
    MUTUAL2
  end;

  procedure MUTUAL2;
  begin
    MUTUAL3
  end;

  procedure MUTUAL3;
  begin
    MUTUAL1
  end;

begin
  MUTUAL1
end.
```

The error report given is

```
Subprogram 'MUTUAL1' has exceeded its recursive call limit of 10000.
The last cycle of mutually recursive calls involves the following
user-defined subprograms:
        MUTUAL1   <-- end of cycle
        MUTUAL3
        MUTUAL2
        MUTUAL1   <-- start of cycle
```

3.3. Parameter Usage Checking

The last example that we shall present is a code auditing tool. The tool is intended to help programmers diagnose certain problems resulting from the misuse of parameters. Data parameters (as opposed to procedural parameters) in Pascal are either passed by reference or passed by value. Reference parameters are used primarily as means for communication between the calling and the called subprogram. Thus, upon subprogram exit, one would normally expect that the subprogram has attributed a value to each of its formal reference parameters. If such is not the case, it could mean either that the called subprogram has failed to return an expected result or that the reference parameter should have been declared as a value parameter in the first place. To help detect these mistakes, the following rules on the usage of reference parameters are prescribed:

1. Each formal reference parameter of a subprogram must be attributed a value during the execution of the subprogram. In other words, each reference parameter must have been given a value by the subprogram when the subprogram terminates.

2. If a reference parameter is a structured object, then an attribution of value to any of its components shall suffice.

3. An exception is made to reference parameters of an array type because in Pascal, read-only array arguments are often passed as reference parameters in order to avoid the run-time overhead in parameter binding.

To enforce these rules, run-time monitoring is necessary. This is so because the presence of assignments to reference parameters does not necessarily imply that such statements are reachable.

Rules on the usage of value parameters are as follows:

1. A value parameter cannot be used as the target of an assignment statement.

2. A value parameter cannot be passed in turn as a reference parameter in subprogram calls, including calls to the standard I/O routines READ and READLN.

The first rule prohibits direct assignments to value parameters, whereas the second rule eliminates the possibility of indirect assignments. Essentially, value parameters are treated as local constants which can be read but not written. Violation of these rules can be detected statically.

There are several rationale behind the above rules. With respect to debugging, a violation of these rules could indicate that a reference parameter has been inadvertently declared as a value parameter. As a style of programming, it has been argued in [13] that using value parameters as temporary variables is a poor programming practice. Finally, incorporating these rules into a language (as is the case in Ada) will obviate the need for the user to reluctantly pass read-only objects as reference parameters in order to gain efficiency; the compiler may choose to simulate pass-by-value using pass-by-reference based on the type of the parameter in question.

We now describe the operation of a tool which enforces the above parameter usage rules. Given a program, the tool first locates:

1. assignment statements whose target is a formal parameter, and

2. subprogram calls where a formal parameter is being passed in turn as a reference parameter.

This information is then used to generate a static diagnostic report and to guide the

insertion of monitor statements. As noted earlier, run-time monitoring is needed for reference parameters only. Instrumentation of the program involves the use of monitor variables to keep track of whether assignments to reference parameters have taken place.

Consider the following program:

```
        program EXAMPLE(INPUT,OUTPUT);
        var X : INTEGER;

          procedure P1(var A : integer; B : CHAR);

            procedure P2(var C : CHAR);
            begin
1             WRITELN(A)
            end;

            procedure P3(D : CHAR);
            begin
1             A := ORD(D) - ORD('0')
            end;

          begin
1           READ(B);
2           A := ORD(B);
3           P2(A);
4           P3(B)
          end;

        begin
1         P1(X,'9')
        end.
```

The instrumented program is presented in Appendix B.3. The static diagnostic report for the above program is given below:

```
Diagnostics on Reference Parameter:
  Subprogram 'P2' -
    'C' is never assigned.

Diagnostics on Value Parameter:
  Subprogram 'P1' -
    'B' is passed as a reference parameter in Subprogram 'P1' statement(s) 1.
```

Executing the instrumented program will produce the following diagnostics:

```
The reference parameter 'C' of Subprogram 'P2' has not been assigned a
value upon subprogram exit.
```

CHAPTER 4

AN ASSESSMENT

Our claim is that the toolkit approach to debugging system design leads to better engineered debugging products. In support of this claim, we shall evaluate our approach against the approach of the prevailing interactive debuggers with respect to some qualitative characteristics of software.

Effectiveness, that is, the efficiency with which errors can be diagnosed, is perhaps the most important attribute of a debugging tool. In this respect, a toolkit system is superior to interactive debuggers when applied to the error domains that the system encompasses. Debugging with interactive debuggers takes place at the execution model level. It entails the understanding of the dynamic behavior of programs through data captured during execution. The vast amount of execution details that must be coped with and comprehended can easily cause the source of errors to be overlooked. This, coupled with the tendency of using the program monitoring facilities in an unstructured manner, would prolong the debugging process unnecessarily. In contrast, the debugging functions offered by a toolkit system are more abstract, and they support the error diagnosis task directly. Instead of tracing the program, the user selects the appropriate tool and the system will verify his suspicion about a certain type of error. The error report produced by the system is more helpful for debugging than the execution state information obtained by tracing. Chances of clerical error are also greatly reduced. Furthermore, systematic debugging is better supported. Even though the support is rather loose, it is a step forward in that

direction, particularly in view of the crude form of current debugging methodologies. All these factors contribute to better debugging productivity.

Simplicity of use is another important attribute in favor of toolkit systems. Using toolkit systems involves selecting the desired tools and possibly answering some straightforward queries. The same procedure applies for accessing all facilities of the system, even if new tools are added. During interaction with the system, the action of the user is directed by the system; the desired user response is clear from the context. The intuitive nature of toolkit systems makes them very easy to use. The use of interactive debuggers, on the other hand, requires learning a command language that bears little resemblance to the source language. The rich set of facilities offered is often reflected in notational complexity. Increasing the power of the debugger will likely make the command language even harder to learn and comprehend. Furthermore interactive debuggers are passive tools; the user must decide on the course of action and issue the proper commands. The lack of directions from the debugger is a disadvantage compared with toolkit systems. Although the wide audience of interactive debuggers is evidence that their complexity can be tolerated, toolkit systems are clearly better alternatives from the human engineering standpoint.

Portability is another measure of quality software. Interactive debuggers are typically tied to the implementation of language translators so that the required symbolic information about the object program can be acquired for carrying out the source-level debugging commands. Because of the tight coupling to particular language translators, interactive debuggers are not easily portable. In comparison, the toolkit systems as proposed in Section 2.5.1 do not rely on the implementation details of compilers; any compiler that accepts the same language as the toolkit system can be used. Thus, portability is improved over typical interactive debuggers.

Completeness is the final attribute that we shall examine. Completeness of a debugging system refers to how applicable the supplied facility is to the debugging problem in general. Interactive debuggers are preferable to toolkit systems in this respect. Because of their low-level nature, interactive debuggers allow the user to diagnose a very broad class of errors. This generality of use is seriously lacking in toolkit systems. The completeness problem in toolkit system design is twofold. On the one hand, it is inconceivable that the universe of all possible programming errors can be completely and meaningfully characterized. Even if such a characterization is found, it is most likely that the error categories are too general to be of use for designing mechanical detection tools. On the other hand, should a sufficiently well-developed error list be already available, it too will probably be of limited use. This is because such an error list will, of necessity, be exceedingly long and a system that encompasses all the error types in the list will overwhelm the user at the outset, defeating the whole purpose of usability. There appears to be no easy solution to this predicament. It should be noted, however, that the utility of a debugging system is only loosely related to completeness. A toolkit system that allows the painless detection of just a few common errors will well justify its implementation effort. The lack of completeness means that toolkit systems will not fully supplant low-level debuggers, but will rather complement them to achieve a more effective overall debugging facility.

CHAPTER 5

EPILOGUE

We have presented a fresh approach to the design of advanced debugging systems. Our work is motivated by the need for debugging aids that are more effective and easier to use than the ubiquitous interactive debuggers, yet they must also be practical. The notion of special-purpose tools is conceived in response to the first set of constraints, and the implementation techniques proposed reflect the recognition of practicality The consolidation and integration of various ideas into a design framework is the main contribution of this thesis.

The toolkit approach is demonstrated through the design and implementation of three tools. An algorithm for detecting uninitialized variable access is also presented. The algorithm differs from existing data flow algorithms in that the former is concerned with reducing the amount of variable usage information collected. Reducing the amount of variable usage information leads in turn to a more concise error report and a reduction in the amount of monitor statements. The preliminary statistics obtained show that the algorithm is able to reduce the total amount of run-time checks by over 50%.

A direction to pursue at this point is to further the implementation effort to produce an experimental toolkit system. An important question that we do not have a satisfactory answer for is the usefulness of toolkit systems. An experimental system will allow us to evaluate the toolkit approach more critically and will certainly bring the deficiencies of our ideas to light.

APPENDIX A

ALGORITHM FOR DETECTING UNINITIALIZED VARIABLE ACCESS

This appendix presents the algorithm for uninitialized variable access detection. As mentioned in Section 3.1, the algorithm consists of an exhaustive method and a supplementary procedure, called the Reduction Algorithm. The exhaustive method is described in the next section. Section A.2 describes the Reduction Algorithm.

A.1. Exhaustive Monitoring

The most obvious method of detecting uninitialized variable access is to exhaustively monitor the program for such errors. To do this, we employ the program instrumentation techniques discussed in Section 2.5.2. First of all, associate a monitor variable with each variable in V, where V is the set of variables in the program for which monitoring is desired. Since we are only interested in knowing whether or not a variable has a valid value, possible values that monitor variables may take are *Initialized* or *NotInitialized*. (That is, the monitor variables used here are *bistate*.) For convenience, monitor variables are declared as BOOLEAN type with TRUE denoting *NotInitialized*. Names for monitor variables are formed by adding the suffix "_UVA" to the original names. For instance, the monitor variable associated with variable "X" is named "X_UVA". Now, monitor statements can be inserted. The types of monitor statements used here are unconditional state-assignment and conditional error-report statement. They are inserted into the program as follows. Let v be a variable in V.

1. For each assignment or input to *v*, insert an unconditional state-assignment of the form

   *v*_UVA := FALSE;

   after the assignment or input statement. This ensures that when a variable receives a valid value, the associated monitor variable is updated accordingly.

2. For each variable access to *v*, insert a conditional error-report statement of the form

   if *v*_UVA **then** *report error*;

   before the statement in which the variable access is part of. This ensures that every variable access is preceded by a validity check.

Of course, declarations of monitor variables and statements to initialize them must also be inserted.

Although this exhaustive algorithm lacks elegance and is rather expensive in terms of execution overhead, it is straightforward and simple to implement. It is used here to handle structured variables (arrays, records, and heap objects) and arbitrary gotos. As mentioned earlier, arrays and heap objects are very difficult to handle statically. We believe the exhaustive algorithm is an effective way to handle them. A more elaborate analysis might actually be more expensive since savings are not always possible with arrays and heap objects. Consider the following example.

```
NEW(P,Q);
Q↑.INFO := SOMETHING;
for I:=X to Y do
  begin
    P↑.INFO := Q↑.INFO;    {error when X < Y}
    NEW(Q)
  end;
P↑.INFO := Q↑.INFO;        {error when X = Y}
```

Here, exhaustive monitoring for heap objects is necessary unless we know the loop

will never execute. It is unlikely that X will always be greater than Y, if that can be determined at all, and the cost incurred in the determination is wasted.

There is a simple extension to the exhaustive algorithm that is worthwhile considering. When an array is being used as a structured constant, it is common to initialize it using a **for**-loop, where the initial and final values of the loop are specified by the constant array bounds. To detect this case, check whether the first operation on an array variable is an assignment inside a **for**-loop and whether the initial and final values of the loop coincide with the array bounds. If so, monitor statements for that array variable are not necessary.

A.2. The Reduction Algorithm

We now present an algorithm that handles simple semistatic variables for programs without **goto** statements and procedure calls; extensions to handle **goto** statements and procedure calls are discussed in separate sections. Unlike the exhaustive algorithm, the aim here is to reduce the number of monitor statements.

```
     program ERRORS(INPUT, OUTPUT);
     var A, B, C, D, X, Y, Z : INTEGER;
     begin
1     X := A + 100;
2     if X > A * D then
3       Y := B + C;
      else
        begin
4         Y := A + C;
5         Z := C + D
        end
6     X := A + B + C + D + Y + Z
     end.
```

Figure A-1: Example to Illustrate the Reduction in Monitor Statements

Consider the insertion of monitor statements for the program in Figure A-1. The goal is to reduce the number of monitor statements needed without leaving any significant error undetected. An error is significant if the same error may not have occurred earlier (and therefore has not necessarily been reported). We will examine the execution effect of the statements and collect monitoring information for the variables. Statements in the program are numbered for convenience of reference.

In statement 1, X is assigned and A is referenced. Since A has not been initialized, the reference to A is a definite error. To avoid generating insignificant diagnostics on A (and save monitor statements), further monitoring on A is not taken. For the same reason, X can also be ignored from now on, even though the value assigned to X is meaningless. In general, if a variable v has been assigned or referenced, then within the range of statements that this is true, all subsequent assignment or reference to v can be ignored. (If the earlier reference is valid, then all subsequent references within that range of statements will be valid. If the earlier reference is invalid, then an error has already been reported and no further error should be generated on the same variable.) Now, monitoring information needs to be collected for only five of the original seven variables. In statement 2, the expression of the **if**-statement gets evaluated first. The reference to D is a definite error. Variable D can now be ignored. Either the then-part or the else-part will be executed next. Even though we do not know which part will be executed next, examination of both branches of the **if**-statement reveals that

1. no matter which part gets executed, Y will be assigned and an error should be generated for the reference to C in either statement 3 or statement 4 (but not statement 5),

2. if the then-part is executed, an error should be generated for the reference to B in statement 3, and

3. if the else-part is executed, Z will be assigned.

The erroneous references mentioned above are definite errors. Since variables C, D, and Y will be either assigned or referenced after the execution of the **if**-statement, we conclude that only B and Z need further monitoring information. In statement 6, the reference to B is a definite error and the reference to Z is a potential error (rather than a definite error because Z will have a value if the else-part of the **if**-statement is executed). To summarize, we have the following information:

definite error	potential error	significant assignment
A: 1	Z: 6	Z: 5
B: 3, 6		
C: 3, 4		
D: 2		
X: 1		

Monitor statements can be inserted based on this information. Definite and potential errors are monitored using, respectively, unconditional and conditional error-report statements; significant assignments are monitored using unconditional state-assignment statements. The statement numbers associated with each variable indicate the relative positions in which to insert the monitor statements. Note that in addition to the initialization monitor statements, nine monitor statements are needed to properly monitor the program. None of them can be eliminated.

From the foregoing discussion, we see that monitoring information for a statement can be obtained by analyzing the execution effect of the statement with respect to assignment and reference operations. The analysis process is repeated for each statement in the program. Monitoring information obtained from each statement is combined together, and, when all statements have been analyzed, the resulting monitoring information is used to guide the insertion of monitor statements. The set of variables for which (partial or complete) monitoring information has been obtained is called the set of *instrumentation variables*, or simply *instrumentation set*.

For each instrumentation variable v, the monitoring information consists of lists of *statement identifications*. A statement identification serves to uniquely identify a statement in the program; statement numbers are used for this purpose in our examples. Each list associated with v represents statements in the program whose execution produces the same effect on v. The relevant execution effect of a statement can be captured using four operations, namely, *definite assignment*, *potential assignment*, *definite reference*, and *potential reference*. A definite (respectively, potential) assignment to variable v occurs if after the execution of a statement, v will (respectively, may) be assigned a value. A definite (respectively, potential) reference to variable v occurs if the execution of a statement may use the value of v, but v is (respectively, may be) uninitialized. For instance, the statement

```
if A > 0 then
  begin
    if A mod 2 = 0 then
      X := 0;
    Y := X
  end
else
  Y := A
```

causes a definite assignment to Y, a potential assignment to X, a definite reference to A, and a potential reference to X. Note that the potential operations are used to express uncertainty about the execution effect of a statement because of the presence of branches. Except for the list of definite-assignment operations, the other three lists are used to guide the insertion of monitor statements. The definite-reference list corresponds to definite errors; the potential-reference list and the potential-assignment list correspond, respectively, to potential errors and significant assignments.

In order to reduce the amount of monitoring information collected (and thereby reduce the number of monitor statements inserted), we keep track of the set of

variables which require further monitoring information. This set is called the set of *active variables*, or simply *active set*. Initially, the active set is the set of variables in the program for which monitoring is desired. In addition to gathering monitoring information for the active variables, the analysis of a statement also determines an updated active set by eliminating those variables which have been definitely assigned or referenced. This updated active set is used in the analysis of succeeding statements.

The term *analysis state* is used to refer to the collective information obtained from the analysis of a statement and other executable language constructs. (An executable construct is a program fragment which specifies a computation on its own. An expression, a list of statements, and a procedure or function are examples of executable constructs, but the label of a statement or the variable on the left hand side of an assignment statement are not.) Specifically, an analysis state contains an active set, an instrumentation set, and monitoring information for each instrumentation variable. Formally, an analysis state, denoted by S, is represented as a two-tuple (A, V), where A is the (updated) active set and V is the instrumentation set. Associated with each variable v in V are four lists, $DA(v)$, $PA(v)$, $DR(v)$, and $PR(v)$, representing, respectively, the definite-assignment list, the potential-assignment list, the definite-reference list, and the potential-reference list.

The problem of instrumenting a program for detecting uninitialized variable access can now be phrased as

> Given a program P consisting of a list of statements L and an active set A for P, compute the analysis state of L with respect to A.

That is, the problem is to determine how the analysis state of different types of statements can be computed and how the analysis states of the statements can be combined together.

```
<statement> ::=
    <assignment>      | <if-then-statement> | <while-loop>
  | <repeat-loop>     | <for-loop>         | <if-statement>
  | <case-statement>  | <with-statement>   | <compound-statement>
  | <null-statement>  | <goto-statement>   | <procedure-call>
<assignment> ::= [<label> :] <variable> := <expression>
<if-then-statement> ::= [<label> :] if <expression> then <statement>
<while-loop> ::= [<label> :] while <expression> do <statement>
<repeat-loop> ::= [<label> :] repeat <statement-list> until <expression>
<for-loop> ::= [<label> :] for <identifier> := <expression1>
  <sequencing-specifier> <expression2> do <statement>
<sequencing-specifier> ::= to | downto
<if-statement> ::=
  [<label> :] if <expression> then <statement1> else <statement2>
<case-statement> ::= [<label> :] case <expression> of <case-clause-list> end
<case-clause-list> ::= <case-clause> {; <case-clause>}
<case-clause> ::= <constant-list> : <statement>
<with-statement> ::= [<label> :] with <variable-list> do <statement>
<compound-statement> ::= [<label> :] begin <statement-list> end
<null-statement> ::= [<label> :]
<goto-statement> ::= [<label> :] goto <target-label>
<procedure-call> ::= [<label> :] <identifier> [( <expression-list> )]
<statement-list> ::= <statement> {; <statement>}
<expression-list> ::= <expression> {, <expression>}
<variable-list> ::= <variable> {, <variable>}
<indexed-variable> ::= <variable> '[' <expression-list> ']'
```

Figure A-2: Syntax of Pascal Statements

The algorithm to be discussed works on the parse-tree representation of programs instead of the flow-graph representation (See Section 2.5.1). The exact structure of the parse tree of a program depends on the grammar used. The grammar of Pascal statements used in our discussion is given in Figure A-2. Although the grammar is incomplete and ambiguous, it suffices to illustrate the basic concepts involved. Several operations on the parse tree are used in our discussion. A selection operation on a node *n* of a parse tree corresponds to moving the focus of attention from that node to another node in the parse tree whose location will be evident from the grammar production corresponding to *n* and from the name of the particular operation used. For instance,

ExpressionOf(node)

where *node* is an assignment statement, selects the rightmost child of *node*. The function *NodeType* takes a node as argument and returns the name of the construct represented by that node. The function *GetStatementIdentification* returns the statement identification of its argument; if the argument is not a statement, it returns the identification of the closest-enclosing statement which contains the argument as a component. The function *MakeExpressionList* is self-explanatory. Other parse tree operations will be explained where appropriate.

Operations on sets used are *set difference*, *union*, and *intersection*. When a new element v is added to the instrumentation set V of an analysis state, it is assumed that the lists associated with v will be automatically set to **nil**, the empty list. When an element is removed from V, its associated lists will no longer be part of the analysis state. Operations on lists are *Append* and *Concat*. *Append* builds a list by destructively appending the list arguments in the given order, assigns the resulting list to the first argument, and sets the rest of the arguments to **nil**. *Concat* takes as arguments an element and a list and returns a new list that is the concatenation of the arguments.

Figure A-3 presents an algorithm to compute the analysis state of different types of Pascal statements, except the **goto** statement and procedure call. In the algorithm, analysis states and their components are subscripted to differentiate them from other program variables.

The algorithm is basically a recursive depth-first-search procedure. In the absence of **goto** statements, visiting nodes in the order indicated resembles closely the actual execution flow. As each node is visited, its analysis state is computed. The context

```
function Analyze(node; A0) return AnalysisState is
  /* node = the program fragment to be analyzed  */
  /* A0 = the active set for node                */
begin
  case NodeType(node) of
    Expression, ExpressionList:
      S1 := (A0, φ)
      W := (the set of simple variables in the expressions) ∩ A0
      RecordReference(S1, W, GetStatementIdentification(node))
      return S1
    Assignment:
      S1 := Analyze(ExpressionOf(node), A0)
      v := VariableOf(node)
      if v is in A0 and is a simple variable then
        RecordAssignment(S1, {v}, GetStatementIdentification(node))
      elsif v is an indexed variable then
        S2 := Analyze(ExpressionListOf(v), A1)
        Combine(S1, S2)
      fi
      return S1
    IfThenStatement, WhileLoop:
      S1 := Analyze(ExpressionOf(node), A0)
      S2 := Analyze(StatementOf(node), A1)
      CombineAlternatives(S2, (A1, φ))
      Combine(S1, S2)
      return S1
    RepeatLoop:
      S1 := Analyze(StatementListOf(node), A0)
      S2 := Analyze(ExpressionOf(node), A1)
      Combine(S1, S2)
      return S1
    ForLoop:
      S1 := Analyze(MakeExpressionList(Expression1Of(node), Expression2Of(node)), A0)
      v := IdentifierOf(node)
      RecordAssignment(S1, {v}, GetStatementIdentification(StatementOf(node)))
      S2 := Analyze(StatementOf(node), A1)
      if not ProvablyExecutable(node) then
        CombineAlternatives(S2, (A1, φ))
      fi
```

Figure A-3: Computing the Analysis State of Statements

```
        A1 := A1 ∪ {v}
        V1 := V1 − {v}
        Combine(S1, S2)
        return S1
    IfStatement:
        S1 := Analyze(ExpressionOf(node), A0)
        S2 := Analyze(Statement1Of(node), A1)
        S3 := Analyze(Statement2Of(node), A1)
        CombineAlternatives(S2, S3)
        Combine(S1, S2)
        return S1
    CaseStatement:
        S1 := Analyze(ExpressionOf(node), A0)
        S2 := Analyze(StatementOf(FirstCaseClauseOf(node)), A1)
        for each remaining case-clause c do
            S3 := Analyze(StatementOf(c), A1)
            CombineAlternatives(S2, S3)
        od
        Combine(S1, S2)
        return S1
    WithStatement:
        /* update scope information */
        return Analyze(StatementOf(node), A0)
    CompoundStatement:
        return Analyze(StatementListOf(node), A0)
    NullStatement:
        return (A0, φ)
    StatementList:
        Stmt := first statement in the list
        S1 := Analyze(Stmt, A0)
        for each remaining statement s in the list do
            S2 := Analyze(s, A1)
            Combine(S1, S2)
        od
        return S1
  esac
end
```

Figure A-3, continued

of a node is immaterial in computing its analysis state; that is, the analysis method used for a node is independent of the surrounding nodes. A consequence of this is that the analysis state of an expression or an assignment statement can be determined easily. For an expression, the analysis state simply contains a record of definite-reference operations for active variables appearing in the expression. The active set and instrumentation set of the analysis state are constructed accordingly. For an assignment statement, its analysis state is obtained by updating the analysis state of the expression of the assignment statement to include a definite-assignment operation for the variable to be assigned. The update is necessary only when the variable to be assigned is active. For example, the analysis state of

X := X + Y + Z

with respect to the active set $A_0 = \{W, X, Y\}$ is

$S_1 = (A_1 = \{W\},\ V_1 = \{X, Y\})$
$DA_1(X) = [1],\ DR_1(X) = [1],\ DR_1(Y) = [1]$

Only nonempty lists are shown. Lists are delimited by square brackets.

The analysis state of other types of statements is determined by recursively computing the analysis states of the executable components of the statement. These analysis states of the components are then manipulated according to the semantics of that statement to obtain the final analysis state.

Intermediate analysis states are manipulated with the auxiliary procedures given in Section A.2.3.2 on Page 85. Procedure *RecordReference* is responsible for updating the analysis state when active variables are being referenced. *RecordAssignment* is the dual of *RecordReference*; it updates the analysis state to indicate assignments to active variables are being taken. Function *ProvablyExecutable* performs a static check to

determine whether a **for**-loop will iterate at least once. It is used in the analysis of **for**-loops in an attempt to obtain stronger analysis information. Two procedures, *Combine* and *CombineAlternatives*, are used for combining analysis states together. *Combine* is used when the analysis states represent program fragments that will be executed in succession. *CombineAlternatives* is used when the analysis states represent program fragments that will be executed in mutual exclusion. Some context information is recovered when analysis state are combined together.

In *CombineAlternatives*, a definite operation will be changed to the corresponding potential operation if the same operation does not occur in both alternatives. Intuitively, this means that if a statement contains branches, we can be sure that a definite operation occurs only if the same operation is performed in each of the branches. For example, in the statement

```
if C then
  begin
    X := 1;
    Y := 2
  end
else
  X := 3;
```

a definite-assignment to X occurs because X is assigned in both branches, but the assignment to Y is only potential.

In *Combine*, redundant potential-assignments are first eliminated. This happens when the potential-assignments occur before a definite-assignment and there are no intervening references, as in

```
if C then
  X := 1;
X := 2;
```

In this example, the first assignment to X does not need to be monitored because no references will depend on it.

We now apply the algorithm to the program in Figure A-1. Let $A_0 = \{A, B, C, D, X, Y, Z\}$ be the initial active set. The analysis state of statement 1 with respect to A_0 is

$$S_1 = (A_1 = \{B,C,D,X,Y,Z\},\ V_1 = \{A,X\})$$
$$DR_1(A) = [1],\ DA_1(X) = [1]$$

The analysis state of the expression of the **if**-statement with respect to A_1 is

$$S_2 = (A_2 = \{B, C, Y, Z\},\ V_2 = \{D\})$$
$$DR_2(D) = [2]$$

The analysis states of the then-part and else-part with respect to A_2 are respectively

$$S_3 = (A_3 = \{Z\},\ V_3 = \{B, C, Y\})$$
$$DR_3(B) = [3],\ DR_3(C) = [3],\ DA_3(Y) = [3]$$

and

$$S_4 = (A_4 = \{B\},\ V_4 = \{C, Y, Z\})$$
$$DR_4(C) = [4],\ DA_4(Y) = [4],\ DA_4(Z) = [5]$$

Combining S_3 and S_4 using *CombineAlternatives* gives the analysis state of the action of the **if**-statement.

$$S_5 = (A_5 = \{B, Z\},\ V_5 = \{B, C, Y, Z\})$$
$$PR_5(B) = [3],\ DR_5(C) = [3, 4],\ DA_5(Y) = [3, 4],\ PA_5(Z) = [5]$$

Combining S_2 and S_5 using *Combine* gives the the analysis state of the **if**-statement.

$$S_6 = (A_6 = \{B, Z\},\ V_6 = \{B, C, D, Y, Z\})$$
$$PR_6(B) = [3],\ DR_6(C) = [3, 4],\ DR_6(D) = [2],\ DA_6(Y) = [3, 4],$$
$$PA_6(Z) = [5]$$

Combining S_1 and S_6 using *Combine*, we obtain the analysis state for the first two statements in the program.

$S_7 = (A_7 = \{B, Z\},\ V_7 = \{A, B, C, D, X, Y, Z\})$
$DR_7(A) = [1],\ PR_7(B) = [3],\ DR_7(C) = [3, 4],\ DR_7(D) = [2],$
$DR_7(X) = [1],\ DA_7(Y) = [3, 4],\ PA_7(Z) = [5]$

The analysis state of statement 6 with respect to A_7 is

$S_8 = (A_8 = \{\},\ V_8 = \{B, Z\})$
$DR_8(B) = [6],\ DR_8(Z) = [6]$

Finally, combining S_7 and S_8 using *Combine*, we obtain

$S_9 = (A_9 = \{\},\ V_9 = \{A, B, C, D, X, Y, Z\})$
$DR_9(A) = [1],\ DR_9(B) = [6],\ PR_9(B) = [3],\ DR_9(C) = [3, 4],$
$DR_9(D) = [2],\ DA_9(X) = [1],\ DA_9(Y) = [3, 4],\ PR_9(Z) = [6],$
$PA_9(Z) = [5]$

The final analysis state indicates that Y has a potential-reference, but no potential-assignment. Since without an assignment, a reference, no matter where it occurs, is always erroneous, the potential-reference must be changed to a definite-reference. The necessary checks for this condition is incorporated in the final version of the algorithm, given in Section A.2.3.

A.2.1. Handling Goto Statements

The rich set of control structures available in Pascal has greatly reduced the need for **goto** statements. The most notable uses of **gotos** in Pascal are for simulating "exits" and "returns". (Incidentally, these are the only forms of explicit jumps provided by Modula2 [70], the successor of Pascal.) We shall extend our algorithm to handle these two important cases. Programs that contain general **gotos** are handled using the exhaustive method. This mixed strategy to handle **gotos** is chosen because we feel that general **gotos** occur very rarely in practice and the use of a sophisticated algorithm will not be cost-effective.

The kinds of **goto** statements that are considered can be characterized as *forward jumps*. A forward jump is a **goto** statement that satisfies the following properties.

1. Both the **goto** and its target appear in the same procedure.
2. The **goto** appears textually before its target.
3. The **goto**, if executed, will terminate the action of the statement list that the **goto** is part of.

(Of course, restrictions set by the language definition must also be observed. See [5].) Forward jumps are illustrated in the following program.

```
program JUMPS(INPUT,OUTPUT);
label 111, 222, 333;
begin
  for I:=1 to 10000 do
    begin
      { some action 1  }
      if C1 then goto 111;   { normal exit }
      { some action 2  }
      if C2 then goto 222;   { long exit   }
    end;
  111: { some action 3  };
  222: { some action 4  };
  if C3 then goto 333;       { return      }
  { some action 5       }
  333:
end;
```

An important property of forward jumps is that if program statements are examined in their textual order, then all jumps to the same label will be encountered before their target.

In computing the analysis state of a statement, our algorithm requires an active set as input. Thus far, the input active set of a statement is taken from the analysis state of the preceding statement. (Except for the first statement in the procedure, in which case the required active set is assumed to be available.) This is valid because

without **goto** statements, the predecessor of a statement (if there is one) is unique. When **gotos** are used, their targets can be reached from more than one point. The problem now is to determine the proper active set for each distinct target label based on the active sets from the **goto** statements.

Our solution involves extending the analysis state to include an additional set *G*. *G* contains the set of target labels encountered thus far. Associate with each label *g* in *G* is an active set *GI(g)*. (*GI* stands for *Goto Information.*) The necessary modification to the *Analyze* procedure is shown in Figure A-4.

A.2.2. Handling Procedure Calls

The last extension that we shall consider is handling procedure and function calls. The modifications to the *Analyze* procedure is presented in Figure A-4. Procedures and functions will be treated synonymously in the following discussion.

The basic idea of our solution is as follows. Each procedure is treated as a separate entity and is analyzed independent of its calling context. Nonlocal variables and reference parameters (but not value parameters) are included in the initial active set for analysis. Now, the previous algorithm can be applied to compute the analysis state of the procedure. When the analysis is completed, the result is stored in a global space for future references. Assume now the analysis states of all procedures have been computed. To handle a procedure call, the appropriate analysis state from the global space is returned, but with information on local variables removed. The analysis state so returned represents the effect of the called procedure on nonlocal entities. The returned analysis state is further shrunk by removing all information that is not of interest at the point of call. The last step is to change all statement identifications from this new analysis state to that of the procedure call, thus effectively hiding the fact that a call has been made.

```
function Analyze(node; A0) return AnalysisState is
begin
  case NodeType(node) of
    Expression, ExpressionList: /* extensions for handling procedure calls */
      S1 := (A0, φ, φ)
      W := (the set of simple variables in the expressions) ∩ A0
      RecordReference(S1, W, GetStatementIdentification(node))
      for each function application f in the expressions do
        S2 := Analyze(f, A1)
        Combine(S1, S2)
      od
      return S1
    ProcedureCall, FunctionApplication:
      S1 := Analyze(ActualValueParametersOf(node), A0)
      p := CalledSubprogramOf(node)
      S2 := Analyze(p, A1)
      Rename(S2, FormalReferenceParametersOf(p),
             ActualReferenceParametersOf(node))
      A2 := A1 − (V2 − A2)
      V2 := V2 ∩ A1
      ChangeStatementIds(S2, GetStatementIdentificaton(node))
      Combine(S1, S2)
      return S1
    Procedure, Function:
      if Visited(node) then
        if AnalysisCompleted(node) then
          S1 := LookUpState(node)
        else
          S1 := (A0, φ, φ)
          AssumeAssignments(S1, {nonlocal variables and reference parameters})
        fi
      else
        Mark(node)  /* mark the node as being visited */
        S1 := Analyze(StatementListOf(node), QueryActiveSet(node))
```

Figure A-4: Handling Goto Statements and Procedure Calls

```
        for each local variable v in V1 do
          if PA1(v) = nil then
            Append(DR1(v), PR1(v))
          else
            Append(PR1(v), DR1(v))
          fi
        od
        InsertState(S1, node)
        /* signal analysis of procedure has been completed */
      fi
      A1 := A1 − {local variables}
      V1 := V1 − {local variables}
      G1 := φ
      return S1
    Assignment:
      .
      (See Figure A-3, page 71)
      .
      .
    StatementList: /* extensions for handling gotos */
      Stmt := first statement in the list
      S1 := Analyze(Stmt, A0)
      for each remaining statement s in the list do
        if s is the target of goto statements then
          UpdateAnalysisState(S1, LabelOf(s))
        fi
        S2 := Analyze(s, A1)
        Combine(S1, S2)
        exit when s is a goto statement
      od
      return S1
    GotoStatement:
      g := TargetLabelOf(node)
      S1 := (A0, φ, {g})
      GI1(g) := A0
      return S1
  esac
end
```

Figure A-4, continued

In computing the analysis state of a procedure, the statements of the procedure may contain recursive calls. This case is handled by making the worse case assumption that all nonlocal variables and reference parameters may be assigned a value. Calls to predefined or library procedures are handled by pre-installing their analysis states in the global space. This implies that predefined or library procedures can only modify the nonlocal environment through the parameters. A last detail worth noting is that insertion of monitor statements is based on the information in the global space and takes place after the entire program has been analyzed.

A.2.3. Complete Listing of Algorithm

A.2.3.1. The Analyze Procedure

```
function Analyze(node; A0) return AnalysisState is
  /* node = the program fragment to be analyzed */
  /* A0 = the active set for node */
begin
  case NodeType(node) of
    Expression, ExpressionList:
      S1 := (A0, φ, φ)
      W := (the set of simple variables in the expressions) ∩ A0
      RecordReference(S1, W, GetStatementIdentification(node))
      for each function application f in the expressions do
        S2 := Analyze(f, A1)
        Combine(S1, S2)
      od
      return S1
    ProcedureCall, FunctionApplication:
      S1 := Analyze(ActualValueParametersOf(node), A0)
      p := CalledSubprogramOf(node)
      S2 := Analyze(p, A1)
      Rename(S2, FormalReferenceParametersOf(p),
             ActualReferenceParametersOf(node))
      A2 := A1 − (V2 − A2)
      V2 := V2 ∩ A1
      ChangeStatementIds(S2, GetStatementIdentificaton(node))
      Combine(S1, S2)
      return S1
    Procedure, Function:
      if Visited(node) then
        if AnalysisCompleted(node) then
          S1 := LookUpState(node)
        else
          S1 := (A0, φ, φ)
          AssumeAssignments(S1, {nonlocal variables and reference parameters}
        fi
      else
        Mark(node) /* mark the node as being visited */
        S1 := Analyze(StatementListOf(node), QueryActiveSet(node))
        for each local variable v̄ in V1 do
          if PA1(v) = nil then
```

```
        Append(DR_1(v), PR_1(v))
      else
        Append(PR_1(v), DR_1(v))
      fi
    od
    InsertState(S_1, node)
    /* signal analysis of procedure has been completed */
  fi
  A_1 := A_1 - {local variables}
  V_1 := V_1 - {local variables}
  G_1 := φ
  return S_1
Assignment:
  S_1 := Analyze(ExpressionOf(node), A_0)
  v := VariableOf(node)
  if v is in A_0 and is a simple variable then
    RecordAssignment(S_1, {v}, GetStatementIdentification(node))
  elsif v is an indexed variable then
    S_2 := Analyze(ExpressionListOf(v), A_1)
    Combine(S_1, S_2)
  fi
  return S_1
IfThenStatement, WhileLoop:
  S_1 := Analyze(ExpressionOf(node), A_0)
  S_2 := Analyze(StatementOf(node), A_1)
  CombineAlternatives(S_2, (A_1, φ, φ))
  Combine(S_1, S_2)
  return S_1
RepeatLoop:
  S_1 := Analyze(StatementListOf(node), A_0)
  S_2 := Analyze(ExpressionOf(node), A_1)
  Combine(S_1, S_2)
  return S_1
ForLoop:
  S_1 := Analyze(MakeExpressionList(Expression1Of(node), Expression2Of(node)), A_0)
  v := IdentifierOf(node)
  RecordAssignment(S_1, {v}, GetStatementIdentification(StatementOf(node)))
  S_2 := Analyze(StatementOf(node), A_1)
  if not ProvablyExecutable(node) then
    CombineAlternatives(S_2, (A_1, φ, φ))
  fi
```

```
    A1 := A1 ∪ {v}
    V1 := V1 − {v}
    Combine(S1, S2)
    return S1
IfStatement:
    S1 := Analyze(ExpressionOf(node), A0)
    S2 := Analyze(Statement1Of(node), A1)
    S3 := Analyze(Statement2Of(node), A1)
    CombineAlternatives(S2, S3)
    Combine(S1, S2)
    return S1
CaseStatement:
    S1 := Analyze(ExpressionOf(node), A0)
    S2 := Analyze(StatementOf(FirstCaseClauseOf(node)), A1)
    for each remaining case-clause c do
        S3 := Analyze(StatementOf(c), A1)
        CombineAlternatives(S2, S3)
    od
    Combine(S1, S2)
    return S1
WithStatement:
    /* update scope information */
    return Analyze(StatementOf(node), A0)
CompoundStatement:
    return Analyze(StatementListOf(node), A0)
NullStatement:
    return (A0, φ, φ)
StatementList:
    Stmt := first statement in the list
    S1 := Analyze(Stmt, A0)
    for each remaining statement s in the list do
        if s is the target of goto statements then
            UpdateAnalysisState(S1, LabelOf(s))
        fi
        S2 := Analyze(s, A1)
        Combine(S1, S2)
        exit when s is a goto statement
    od
    return S1
GotoStatement:
    g := TargetLabelOf(node)
```

```
      S1 := (A0, φ, {g})
      GI1(g) := A0
      return S1
  esac
end
```

A.2.3.2. Auxiliary Procedures

```
procedure RecordReference(var S0; W; StatementId) is
  /* S0 = (A0, V0, G0), an analysis state                          */
  /* W = a set of variables which are referenced                   */
  /* StatementId = the statement where the reference operations occur */
begin
  A0 := A0 − W
  V0 := V0 ∪ W
  for each variable v in W do
    DR0(v) := Concat(StatementId, nil)
  od
end

procedure RecordAssignment(var S0; W; StatementId) is
begin
  A0 := A0 − W
  V0 := V0 ∪ W
  for each variable v in W do
    DA0(v) := Concat(StatementId, nil)
  od
end

function ProvablyExecutable(ForLoop) return boolean is
  /* ForLoop = a program fragment which is a for-loop */
begin
  if both loop-expressions are constant expressions then
    Diff := difference between final and initial value of loop
    return (Diff = 0) or (Diff > 0 and IsForToLoop(ForLoop)) or
           (Diff < 0 and IsForDowntoLoop(ForLoop))
  else
    return false
  fi
end

procedure CombineAlternatives(var S1; S2) is
  /* S1, S2 = analysis states of alternatives */
```

```
begin
  A1 := A1 ∪ A2
  for each v in V1 ∪ V2 do
    if v in V1 ∩ V2 then
      if DR1(v) ≠ nil and DR2(v) ≠ nil then
        Append(DR1(v), DR2(v))
      elsif DR1(v) ≠ nil then
        DRtoPR(S1, v)
      elsif DR2(v) ≠ nil then
        DRtoPR(S2, v)
        DR1(v) := DR2(v)
      fi
      if DA1(v) ≠ nil and DA2(v) ≠ nil then
        Append(DA1(v), DA2(v))
      elsif DA1(v) ≠ nil then
        Append(PA1(v), DA1(v))
      elsif DA2(v) ≠ nil then
        Append(PA1(v), DA2(v))
      fi
      Append(PR1(v), PR2(v))
      Append(PA1(v), PA2(v))
    elsif v in V1 then
      DRtoPR(S1, v)
      Append(PA1(v), DA1(v))
    else /* v in V2 − V1 */
      V1 := V1 ∪ {v}
      DRtoPR(S2, v)
      DR1(v) := DR2(v)
      PR1(v) := PR2(v)
      Append(PA1(v), DA2(v), PA2(v))
    fi
  od
  for each g in G2 do /* used to handle gotos */
    if g is in G1 then
      GI1(g) := GI1(g) ∪ GI2(g)
    else /* g in G2 − G1 */
      G1 := G1 ∪ {g}
      GI1(g) := GI2(g)
    fi
  od
```

```
end

procedure Combine(var S1; S2) is
  /* S1, S2 = successive analysis states                                  */
  /* AssignedBeforeRefIn2 = flag to indicate if an active variable        */
  /*       in A1 is assigned before any reference occurs in S2            */
begin
  A1 := A2
  for each v in V2 do
    AssignedBeforeRefIn2 := (DA2(v) ≠ nil and PR2(v) = nil)
    if AssignedBeforeRefIn2 then
      PA2(v) := nil
    fi
    if v in V1 then
      if AssignedBeforeRefIn2 and PR1(v) = nil then
        PA1(v) := nil
      fi
      Append(DA1(v), DA2(v))
      if PA1(v) = nil then
        Append(DR1(v), DR2(v))
        Append(PR1(v), PR2(v))
        PA1(v) := PA2(v)
      else
        Append(PR1(v), DR2(v), PR2(v))
        Append(PA1(v), PA2(v))
      fi
    else /* v in V2 - V1 */
      V1 := V1 ∪ {v}
      DA1(v) := DA2(v)
      DR1(v) := DR2(v)
      PR1(v) := PR2(v)
      PA1(v) := PA2(v)
    fi
  od
  for each g in G2 - G1 do /* used to handle gotos */
    G1 := G1 ∪ {g}
    GI1(g) := GI2(g)
  od
end

procedure UpdateAnalysisState(var S0; g) is
```

```
begin
  for each v in GI0(g) − A0 do
    Append(PA0(v), DA0(v))
  od
  A0 := GI0(g)
  G0 := G0 − {g}
end

procedure ~~ChangeStatementIds~~(var S0; ReplacementId) is
  /* S0 = (A0, V0, G0), an analysis state */
  /* ReplacementId = new identification */
begin
  for each v in V0 do
    if DA0(v) ≠ nil then
      DA0(v) := Concat(ReplacementId, nil)
    fi
    if PA0(v) ≠ nil then
      PA0(v) := Concat(ReplacementId, nil)
    fi
    if DR0(v) ≠ nil then
      DR0(v) := Concat(ReplacementId, nil)
    fi
    if PR0(v) ≠ nil then
      PR0(v) := Concat(ReplacementId, nil)
    fi
  od
end

procedure AssumeAssignments(var S0; W) is
  /* DummyId = a unique nonexisting statement id, ignored during instrumentation */
begin
  V0 := V0 ∪ W
  for each variable v in W do
    PA0(v) := Append(PA0(v),Concat(DummyId, nil))
  od
end
```

APPENDIX B

LISTING OF INSTRUMENTED PROGRAMS

This appendix presents the instrumented programs for the examples given in Chapter 3.

B.1. Uninitialized Variable Access Detection

```
PROGRAM EXAMPLE (INPUT, OUTPUT);

  TYPE
    LIST = ↑ NODE;
    NODE =
      RECORD
        NUMUVA : boolean; LINKUVA : boolean; NUM : INTEGER; LINK : LIST
      END;

  VAR
    NUMBERS, ENTRY : LIST;
    FOUND, LAST : BOOLEAN;
    NEWNUM : INTEGER;
    ErrorFile : text;
    UVANameMap : PACKED ARRAY [1..2, 1..7] OF char;
    LASTUVA : boolean;
    NEWNUMUVA : boolean;


  PROCEDURE UVAerr1 (VAR MonVar : boolean;
                     NameIndex, StatementNumber : integer);

    VAR
      I : integer;

  BEGIN
    WRITE (
```

```
    ErrorFile, '    Accessing uninitialized variable at statement ',
    StatementNumber : 0, ' of subprogram "');
  FOR I := 1 TO 7 DO
    IF UVANameMap[NameIndex, I] <> ' ' THEN
      WRITE (ErrorFile, UVANameMap[NameIndex, I]);
  WRITELN (ErrorFile, '".');
    { When a full report is selected, the tool generates
      "MonVar := false" in place of the following code    };
  WRITELN (ErrorFile);
  WRITELN (ErrorFile, 'Program execution is halted.');
  halt
END;


PROCEDURE UVAerr2 (NameIndex, StatementNumber : integer);

  VAR
    I : integer;

BEGIN
  WRITE (
    ErrorFile, '    Accessing inactive variant field at statement ',
    StatementNumber : 0, ' of subprogram "');
  FOR I := 1 TO 7 DO
    IF UVANameMap[NameIndex, I] <> ' ' THEN
      WRITE (ErrorFile, UVANameMap[NameIndex, I]);
  WRITELN (ErrorFile, '".');
    { When a full report is selected, the tool replaces
      the following code by a null statement             }
  WRITELN (ErrorFile);
  WRITELN (ErrorFile, 'Program execution is halted.');
  halt
END;


PROCEDURE UVAInit;

BEGIN
  rewrite(ErrorFile, '-ERROR ');
  WRITELN (ErrorFile, 'Execution Errors Detected:');
  UVANameMap[1] := 'EXAMPLE';
  UVANameMap[2] := 'SEARCH '
END;
```

```
PROCEDURE NODEUVAs (VAR MonVar : NODE; NewValue : boolean);

BEGIN
  WITH MonVar↑ DO
    BEGIN
      NUMUVA := NewValue; LINKUVA := NewValue
    END
END;


PROCEDURE NODEUVAc (VAR MonVar : NODE; Subprogram, Statement : integer);

BEGIN
  WITH MonVar↑ DO
    BEGIN
      IF NUMUVA THEN UVAerr1(NUMUVA, Subprogram, Statement);
      IF LINKUVA THEN UVAerr1(LINKUVA, Subprogram, Statement)
    END
END;


PROCEDURE SEARCH
            (L : LIST; NUMBER : INTEGER; VAR SUCCESS : BOOLEAN);

  VAR
    CURR : LIST;

BEGIN
  SUCCESS := FALSE;
  CURR := L;
  WHILE CURR <> NIL DO
    BEGIN
      IF CURR↑.NUMUVA THEN UVAerr1(CURR↑.NUMUVA, 2, 4);
      IF CURR↑.NUM = NUMBER THEN
        BEGIN
          SUCCESS := TRUE; CURR := NIL
        END
      ELSE
      BEGIN
        IF CURR↑.LINKUVA THEN UVAerr1(CURR↑.LINKUVA, 2, 7);
        CURR := CURR↑.LINK
      END
    END
END;
```

```
BEGIN
  UVAInit;
  LASTUVA := true;
  NEWNUMUVA := true;
  NUMBERS := NIL;
  REPEAT
    IF NEWNUMUVA THEN UVAerr1(NEWNUMUVA, 1, 3);
    SEARCH(NUMBERS, NEWNUM, FOUND);
    IF FOUND THEN
      BEGIN
        LAST := FALSE; LASTUVA := false
      END
    ELSE
    BEGIN
      NEW(ENTRY);
      NODEUVAs(ENTRY↑, true);
      ENTRY↑.NUM := NEWNUM;
      ENTRY↑.NUMUVA := false;
      ENTRY↑.LINK := NUMBERS;
      ENTRY↑.LINKUVA := false;
      NUMBERS := ENTRY;
      IF LASTUVA THEN UVAerr1(LASTUVA, 1, 10);
      IF LAST THEN WRITELN ('Consecutive new entries.')
      ELSE LAST := TRUE
    END
  UNTIL EOF
END.
```

B.2. Bounded Execution Failure Diagnosis

```
PROGRAM EXAMPLE1 (INPUT, OUTPUT);

  TYPE
    BEFStackType = ↑ BEFStackNode;
    BEFStackNode =
      RECORD
        Info : integer; Link : BEFStackType
      END;

  VAR
    I, J, K : INTEGER;
    BEFLoopCounters : ARRAY [1..3] OF integer { inserted by debugger };
    ErrorFile : text;
    BEFLoopStack : BEFStackType;
```

```
  BEFCallStack : BEFStackType;


PROCEDURE BEFPush (VAR Stack : BEFStackType; ProcNameIndex : integer);

  VAR
    StackElem : BEFStackType;

BEGIN
  new(StackElem);
  StackElem↑.Info := ProcNameIndex;
  StackElem↑.Link := Stack;
  Stack := StackElem
END;


FUNCTION BEFPop (VAR Stack : BEFStackType) : integer;

  VAR
    StackElem : BEFStackType;

BEGIN
  StackElem := Stack;
  Stack := Stack↑.Link;
  BEFPop := StackElem↑.Info;
  dispose(StackElem)
END;


PROCEDURE BEFLoopErr (LoopLineNum : integer; OuterLoops : integer);

  VAR
    LineNumber, Iterations, I : integer;

BEGIN
  WRITELN (sercom, '*** Infinite loop detected.  Execution is halted.');
  WRITELN (sercom, '*** See error report in "-ERROR"');
  WRITE (
    ErrorFile, 'The loop beginning at statement ', LoopLineNum : 0, ' in ');
  WRITE (ErrorFile, '''EXAMPLE1''');
  WRITELN (
    ErrorFile, ''' has exceeded its iteration limit of ', 10000 : 0, '.');
  IF OuterLoops > 1 THEN
    BEGIN
      WRITELN (ErrorFile, 'Iteration status of outer loops:');
```

```
        FOR I := 1 TO OuterLoops DO
          BEGIN
            LineNumber := BEFPop(BEFLoopStack);
            Iterations := BEFPop(BEFLoopStack);
            WRITELN (
              ErrorFile, ' ' : 5, 'Loop at statement ', LineNumber : 0,
              ' has iterated ', Iterations : 0, ' times.')
          END
      END;
    halt
  END;


  PROCEDURE BEFInit;

  BEGIN
    rewrite(ErrorFile, '-ERROR '); BEFLoopStack := NIL
  END;


  PROCEDURE BEFExit;

  BEGIN
    WRITELN (
      sercom, '*** No infinite loop or infinite recursion has been',
      ' detected.');
    WRITELN (sercom);
    WRITELN (ErrorFile, 'All loops iterate less than the limit of ', ' times.');
    WRITELN (
      ErrorFile, 'Recursive subprogram calls have not exceeded',
      ' the depth limit of ', '.')
  END;

BEGIN
  BEFInit;
  J := 0;
  K := 0;
  BEFLoopCounters[1] := 0;
  FOR I := 1 TO 100 DO
    BEGIN
      IF BEFLoopCounters[1] = 10000 THEN BEFLoopErr(3, 0)
      ELSE BEFLoopCounters[1] := BEFLoopCounters[1] + 1;
      IF I = 24 THEN
        BEGIN
          BEFLoopCounters[2] := 0;
```

```
        REPEAT
          IF BEFLoopCounters[2] = 10000 THEN
            BEGIN
              BEFPush(BEFLoopStack, BEFLoopCounters[1]);
              BEFPush(BEFLoopStack, 3);
              BEFLoopErr(5, 1)
            END
          ELSE BEFLoopCounters[2] := BEFLoopCounters[2] + 1;
          J := J + 1;
          IF J = 15 THEN
            BEGIN
              BEFLoopCounters[3] := 0;
              WHILE true DO
                BEGIN
                  IF BEFLoopCounters[3] = 10000 THEN
                    BEGIN
                      BEFPush(BEFLoopStack, BEFLoopCounters[2]);
                      BEFPush(BEFLoopStack, 5);
                      BEFPush(BEFLoopStack, BEFLoopCounters[1]);
                      BEFPush(BEFLoopStack, 3);
                      BEFLoopErr(8, 2)
                    END
                  ELSE BEFLoopCounters[3] := BEFLoopCounters[3] + 1;
                  K := K + 1
                END
            END
        UNTIL FALSE
      END
  END;
BEFExit
END.

PROGRAM EXAMPLE2 (INPUT, OUTPUT);

  TYPE
    BEFStackType = ↑ BEFStackNode;
    BEFStackNode =
      RECORD
        Info : integer; Link : BEFStackType
      END;

  VAR
    ErrorFile : text { inserted by debugger };
    BEFLoopStack : BEFStackType;
    BEFCallStack : BEFStackType;
```

```
  BEFNameMap : PACKED ARRAY [1..1, 1..13] OF char;
  BEFProcCounters : ARRAY [1..1] OF integer;


PROCEDURE BEFPush (VAR Stack : BEFStackType; ProcNameIndex : integer);

  VAR
    StackElem : BEFStackType;

BEGIN
  new(StackElem);
  StackElem↑.Info := ProcNameIndex;
  StackElem↑.Link := Stack;
  Stack := StackElem
END;


PROCEDURE BEFRemoveTopElem (VAR Stack : BEFStackType);

  VAR
    StackElem : BEFStackType;

BEGIN
  StackElem := Stack; Stack := Stack↑.Link; dispose(StackElem)
END;


PROCEDURE BEFCallErr;

  VAR
    Key : integer;
    Next : BEFStackType;

BEGIN
  WRITELN (sercom, '*** Infinite recursion detected.  Execution is halted.');
  WRITELN (sercom, '*** See error report in "-ERROR"');
  Key := BEFCallStack↑.Info;
  Next := BEFCallStack↑.Link;
  WRITELN (
    ErrorFile, 'Subprogram ''', BEFNameMap[Key], ''' has exceeded its ',
    'recursive call limit of ', 10000 : 0, '.');
  IF Next↑.Info = Key THEN
    WRITELN (ErrorFile, 'The subprogram is self-recursive.')
  ELSE
  BEGIN
```

```
    WRITELN (
      ErrorFile, 'The last cycle of mutually recursive calls',
      ' involves the following user-defined subprograms:');
    WRITELN (ErrorFile, ' ' : 5, BEFNameMap[Key], '    <-- end of cycle');
    REPEAT
      WRITELN (ErrorFile, ' ' : 5, BEFNameMap[Next↑.Info]); Next := Next↑.Link
    UNTIL Next↑.Info = Key;
    WRITELN (
      ErrorFile, ' ' : 5, BEFNameMap[Next↑.Info], '    <-- start of cycle')
  END;
  halt
END;


PROCEDURE BEFInit;

  VAR
    I : integer;

BEGIN
  rewrite(ErrorFile, '-ERROR ');
  BEFCallStack := NIL;
  FOR I := 1 TO 1 DO BEFProcCounters[I] := 0;
  BEFNameMap[1] := 'SELFRECURSIVE'
END;


PROCEDURE BEFExit;

BEGIN
  WRITELN (
    sercom, '*** No infinite loop or infinite recursion has been',
    ' detected.');
  WRITELN (sercom);
  WRITELN (ErrorFile, 'All loops iterate less than the limit of ', ' times.');
  WRITELN (
    ErrorFile, 'Recursive subprogram calls have not exceeded',
    ' the depth limit of ', '.')
END;


PROCEDURE SELFRECURSIVE;

BEGIN
  BEFPush(BEFCallStack, 1);
```

```
    IF BEFProcCounters[1] = 10000 THEN BEFCallErr
    ELSE BEFProcCounters[1] := BEFProcCounters[1] + 1;
    SELFRECURSIVE;
    BEFProcCounters[1] := BEFProcCounters[1] - 1;
    BEFRemoveTopElem(BEFCallStack)
  END;

BEGIN
  BEFInit; SELFRECURSIVE; BEFExit
END.

PROGRAM EXAMPLE3 (INPUT, OUTPUT);

  TYPE
    BEFStackType = ↑ BEFStackNode;
    BEFStackNode =
      RECORD
        Info : integer; Link : BEFStackType
      END;

  VAR
    ErrorFile : text { inserted by debugger };
    BEFLoopStack : BEFStackType;
    BEFCallStack : BEFStackType;
    BEFNameMap : PACKED ARRAY [1..3, 1..7] OF char;
    BEFProcCounters : ARRAY [1..3] OF integer;


  PROCEDURE BEFPush (VAR Stack : BEFStackType; ProcNameIndex : integer);

    VAR
      StackElem : BEFStackType;

  BEGIN
    new(StackElem);
    StackElem↑.Info := ProcNameIndex;
    StackElem↑.Link := Stack;
    Stack := StackElem
  END;


  PROCEDURE BEFRemoveTopElem (VAR Stack : BEFStackType);

    VAR
      StackElem : BEFStackType;
```

```
BEGIN
  StackElem := Stack; Stack := Stack↑.Link; dispose(StackElem)
END;


PROCEDURE BEFCallErr;

  VAR
    Key : integer;
    Next : BEFStackType;

BEGIN
  WRITELN (sercom, '*** Infinite recursion detected.  Execution is halted.');
  WRITELN (sercom, '*** See error report in "-ERROR"');
  Key := BEFCallStack↑.Info;
  Next := BEFCallStack↑.Link;
  WRITELN (
    ErrorFile, 'Subprogram "', BEFNameMap[Key], '" has exceeded its ',
    'recursive call limit of ', 10000 : 0, '.');
  IF Next↑.Info = Key THEN
    WRITELN (ErrorFile, 'The subprogram is self-recursive.')
  ELSE
  BEGIN
    WRITELN (
      ErrorFile, 'The last cycle of mutually recursive calls',
      ' involves the following user-defined subprograms:');
    WRITELN (ErrorFile, ' ' : 5, BEFNameMap[Key], '    <-- end of cycle');
    REPEAT
      WRITELN (ErrorFile, ' ' : 5, BEFNameMap[Next↑.Info]); Next := Next↑.Link
    UNTIL Next↑.Info = Key;
    WRITELN (
      ErrorFile, ' ' : 5, BEFNameMap[Next↑.Info], '    <-- start of cycle')
  END;
  halt
END;


PROCEDURE BEFInit;

  VAR
    I : integer;

BEGIN
  rewrite(ErrorFile, '-ERROR ');
  BEFCallStack := NIL;
```

```
  FOR I := 1 TO 3 DO BEFProcCounters[I] := 0;
  BEFNameMap[1] := 'MUTUAL1';
  BEFNameMap[2] := 'MUTUAL2';
  BEFNameMap[3] := 'MUTUAL3'
END;


PROCEDURE BEFExit;

BEGIN
  WRITELN (
    sercom, '*** No infinite loop or infinite recursion has been',
    ' detected.');
  WRITELN (sercom);
  WRITELN (ErrorFile, 'All loops iterate less than the limit of ', ' times.');
  WRITELN (
    ErrorFile, 'Recursive subprogram calls have not exceeded',
    ' the depth limit of ', '.')
END;


PROCEDURE MUTUAL2;
    forward;


PROCEDURE MUTUAL3;
    forward;


PROCEDURE MUTUAL1;

BEGIN
  BEFPush(BEFCallStack, 1);
  IF BEFProcCounters[1] = 10000 THEN BEFCallErr
  ELSE BEFProcCounters[1] := BEFProcCounters[1] + 1;
  MUTUAL2;
  BEFProcCounters[1] := BEFProcCounters[1] - 1;
  BEFRemoveTopElem(BEFCallStack)
END;


PROCEDURE MUTUAL2;

BEGIN
  BEFPush(BEFCallStack, 2);
```

```
    IF BEFProcCounters[2] = 10000 THEN BEFCallErr
    ELSE BEFProcCounters[2] := BEFProcCounters[2] + 1;
    MUTUAL3;
    BEFProcCounters[2] := BEFProcCounters[2] - 1;
    BEFRemoveTopElem(BEFCallStack)
  END;


  PROCEDURE MUTUAL3;

  BEGIN
    BEFPush(BEFCallStack, 3);
    IF BEFProcCounters[3] = 10000 THEN BEFCallErr
    ELSE BEFProcCounters[3] := BEFProcCounters[3] + 1;
    MUTUAL1;
    BEFProcCounters[3] := BEFProcCounters[3] - 1;
    BEFRemoveTopElem(BEFCallStack)
  END;

BEGIN
  BEFInit; MUTUAL1; BEFExit
END.
```

B.3. Parameter Usage Checking

```
PROGRAM EXAMPLE (INPUT, OUTPUT);

  VAR
    X : INTEGER;


  PROCEDURE P1 (VAR A : INTEGER; B : CHAR);

    VAR
      APUC : boolean;


    PROCEDURE P2 (VAR C : INTEGER);

    BEGIN
      WRITELN (A);
      WRITELN (
        sercom, 'The reference parameter ', '''C''', ' of subprogram ', '''P2''',
        ' has not been assigned a value upon subprogram exit.')
```

```
    END;


    PROCEDURE P3 (D : CHAR);

    BEGIN
      A := ORD(D) - ORD('0'); APUC := false
    END;

  BEGIN
    APUC := true;
    READ(B);
    A := ORD(B);
    APUC := false;
    P2(A);
    APUC := false;
    P3(B);
    IF APUC THEN
      WRITELN (
        sercom, 'The reference parameter ', '''A''', ' of subprogram ', '''P1''',
        ' has not been assigned a value upon subprogram exit.')
  END;

BEGIN
  P1(X, '9')
END.
```

REFERENCES

[1] Adam, Ann and Laurent, Jean-Pierre.
LAURA, a system to debug student programs.
Artificial Intelligence 15(1), 1980.

[2] adb.
Commands Reference Manual for the Sun Workstation.
Sun Microsystems, Inc., California, 1985.

[3] Aho, V. Alfred and Ullman, Jeffery D.
Principles of Compiler Design.
Addison-Wesley Publishing Company, Massachusetts, 1977.

[4] Amsterdam, Jonathan.
Programming project: safe storage allocator.
BYTE Magazine 11(10), October, 1986.

[5] *ANSI/IEEE 770 X3.97 - 1983, IEEE Standard Pascal Computer Programming Language*
American National Standards Institute, 1983.

[6] Baiardi, F., De Francesco, N., Matteoli, E., Stefanini, S., and Vaglini, G.
Development of a debugger for a concurrent language.
SIGPLAN Notices 18(8), August, 1983.
(*Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on High-Level Debugging*, March, 1983).

[7] Balzer, R. M.
EXDAMS -- extendable debugging and monitoring system.
AFIPS Conference Proceedings 34, 1969.

[8] Bates, Peter and Wileden, Jack C.
An approach to high-level debugging of distributed systems.
SIGPLAN Notices 18(8), August, 1983.
(*Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on High-Level Debugging*, March, 1983).

[9] Beander, Bert.
VAX DEBUG: an interactive, symbolic, multilingual debugger.
SIGPLAN Notices 18(8), August, 1983.
(*Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on High-Level Debugging*, March, 1983).

[10] Boehm, B. W., McClean, R. K., and Urfrig, D. B.
Some experience with automated aids to the design of large-scale software.
IEEE Transactions on Software Engineering SE-1(2), March, 1975.

[11] Cameron, Robert D. and Ito, M. Robert.
Grammar-based definition of metaprogramming systems.
ACM Transactions on Programming Languages and Systems 6(1), January, 1984.

[12] Cameron, Robert D.
Multi MPS Reference Manual (draft)
School of Computing Science, Simon Fraser University, 1986.

[13] Cameron, Robert D.
Value parameters should be read-only.
Unpublished proposal submitted to the ISO Modula2 Standardization Working Group, 1987.

[14] Cardell, James R.
Multilingual debugging with the SWAT high-level debugger
SIGPLAN Notices 18(8), August, 1983.
(*Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on High-Level Debugging*, March, 1983).

[15] Cargill, Thomas A.
Debugging C programs with the Blit.
AT & T Bell Laboratories Technical Journal 63(8), October, 1984.

[16] Cordell, Green and Barstow, David.
On program synthesis knowledge.
Artificial Intelligence 10(3), November, 1978.

[17] Dahl, O.-J., Dijkstra, E. W., and Hoare, C. A. R.
Structured Programming.
Academic Press, New York, 1972.

[18] Dannenberg, R. B. and Ernest, G. W.
Formal program verification using symbolic execution.
IEEE Transactions on Software Engineering SE-8(1), January, 1982.

[19] dbx.
Commands Reference Manual for the Sun Workstation.
Sun Microsystems, Inc., California, 1985.

[20] dbxtool.
Commands Reference Manual for the Sun Workstation.
Sun Microsystems, Inc., California, 1985.

[21] DeMarco, Tom.
Controlling Software Projects: Management, Measurement, and Estimation.
Yourdon Press, New York, 1982.

[22] Dijkstra, E. W.
Goto statement considered harmful.
Communications of the ACM 11(3), March, 1968.

[23] Endres, Albert.
An analysis of errors and causes in system programs.
SIGPLAN Notices 10(6), June, 1975.
(*Proceedings-1975 International Conference on Reliable Software*, April, 1975).

[24] Fairley, Richard E.
ALADDIN: assembly language assertion driven debugging interpreter.
IEEE Transactions on Software Engineering SE-5(4), July, 1979.

[25] Fairley, Richard E.
Software Engineering Concepts.
McGraw-Hill Book Company, New York, 1985.

[26] Ferguson, Earl H. and Berner, Elizabeth.
Debugging systems at the source language level.
Communications of the ACM 6(8), August, 1963.

[27] Fosdick, D. Lloyd and Osterweil, Leon J.
Data flow analysis in software reliability.
ACM Computing Surveys 8(3), September, 1976.

[28] Foxley, E. and Morgan, D. J.
Monitoring the run-time activity of Algol 68-R programs.
Software Practice & Experience 8(1), January, 1978.

[29] Frenkel, Karen A.
Toward automating the software-development cycle.
Communications of the ACM 28(6), June, 1985.

[30] Graham, S. L., Kessler, P. B., and McKusick, M. K.
An execution profiler for modular programs.
Software Practice & Experience 13(8), August, 1983.

[31] Grishman, Ralph.
The debugging system AIDS.
AFIPS Conference Proceedings 36, 1970.

[32] Gupta, N. K. and Seviora, R. E.
An expert system approach to real time system debugging.
In *Proceedings of the 1st Conference on Artificial Intelligence Applications*. December, 1984.

[33] Hantler, Sidney L. and King, James C.
An introduction to proving correctness of programs.
ACM Computing Surveys 8(3), September, 1976.

[34] Hecht, Matthew S.
Flow Analysis of Computer Programs.
North-Holland Publishing Inc., New York, 1977.

[35] Hoare, C. A. R.
An axiomatic basis for computer programming.
Communications of the ACM 12(10), October, 1969.

[36] Hoare, C. A. R.
Data reliability.
SIGPLAN Notices 10(6), June, 1975.
(*Proceedings-1975 International Conference on Reliable Software*, April, 1975).

[37] Howden, William E.
Symbolic testing and the DISSECT symbolic evaluation system.
IEEE Transactions on Software Engineering SE-3(4), July, 1977.

[38] Howden, William E.
Functional program testing.
IEEE Transactions on Software Engineering SE-6(2), March, 1980.

[39] Huang, J. C.
An approach to program testing.
ACM Computing Surveys 7(3), September, 1975.

[40] Huang, J. C.
Program instrumentation and software testing.
Computer 11(4), April, 1978.

[41] Huang, J. C.
Detection of data flow anomaly through program instrumentation.
IEEE Transactions on Software Engineering SE-5(3), May, 1979.

[42] Itoh, Daiju and Izutani, Takao.
FADEBUG-I, a new tool for program debugging.
In *Proceedings of the IEEE Symposium on Computer Software Reliability*. May, 1973.

[43] Johnson, Mark Scott.
The Design and Implementation of a Run-Time Analysis and Interactive Debugging Environment.
PhD thesis, University of British Columbia, 1978.

[44] Jones, T. C.
Programming Productivity: Issues for the Eighties.
IEEE Computer Society Press, New York, 1981.

[45] Kernighan, Brian W. and Pike, Rob.
The UNIX Programming Environment.
Prentice-Hall, Inc., New Jersey, 1984.

[46] Lichtman, Zavdi L.
Generation and consistency checking of design and program structures.
IEEE Transactions on Software Engineering SE-12(1), January, 1986.

[47] LINT -- a C program checker.
Programming Tools for the Sun Workstation.
Sun Microsystems, Inc., California, 1985.

[48] Manna, Zohar and Waldinger, Richard.
Knowledge and reasoning in program synthesis.
Artificial Intelligence 6(2), Summer, 1975.

[49] Mauger, Claude and Pammett, Kevin.
An event-driven debugger for Ada.
ACM SIGAda Ada Letters 5(2), September, 1985.
(*Proceedings of the Ada International Conference*, May, 1985).

[50] Muchnick, Steven S. and Jones, Neil D.
Program Flow Analysis: Theory and Applications.
Prentice-Hall, Inc., New Jersey, 1981.

[51] Myers, Glenford J.
The Art of Software Testing.
Wiley Interscience, New York, 1979.

[52] Osterweil, Leon J. and Fosdick, Lloyd D.
DAVE -- a validation error detection and documentation system for Fortran programs.
Software Practice & Experience 6(4), October, 1976.

[53] Pai, Ajit B. and Kieburtz, Richard B.
Global context recovery: a new strategy for syntactic error recovery by table-driven parsers.
ACM Transactions on Programming Languages and Systems 2(1), January, 1980.

[54] Popek, G., Horning, J., Lampson, B., Mitchell, J., and London, R.
Notes on the design of Euclid.
SIGPLAN Notices 12(3), March, 1977.

[55] *Reference Manual for the ADA Programming Language, ANSI/MIL-STD-1815A-1983*
United States Department of Defense, 1983.

[56] Reynolds, J. C.
Syntactic control of interference.
In *Proceedings of the Fifth Annual ACM Symposium on Principles of Programming Languages*. January, 1979.

[57] Richter, Helmut.
Noncorrecting syntax error recovery.
ACM Transactions on Programming Languages and Systems 7(3), July, 1985.

[58] Rubey, Raymond L.
Quantitative aspects of software validation.
SIGPLAN Notices 10(6), June, 1975.
(*Proceedings-1975 International Conference on Reliable Software*, April, 1975).

[59] Ruth, Gregory R.
Intelligent program analysis.
Artificial Intelligence 7(1), 1976.

[60] Satterthwaite, E.
Debugging tools for high level languages.
Software Practice & Experience 2(3), July, 1972.

[61] Sedlmeyer, Robert L., Thompson, William B., and Johnson, Paul E.
Knowledge-based fault localization in debugging.
SIGPLAN Notices 18(8), August, 1983.
(*Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on High-Level Debugging*, March, 1983).

[62] Shapiro, Ehud Y.
Algorithmic Program Debugging.
PhD thesis, Yale University, 1982.

[63] Shooman, M. L. and Bolsky, M. I.
Types, distribution, and test and correction times for programming errors.
SIGPLAN Notices 10(6), June, 1975.
(*Proceedings-1975 International Conference on Reliable Software*, April, 1975).

[64] Sippu, Seppo and Soisalon-Soininen, Eljas.
A syntax-error-handling technique and its experimental analysis.
ACM Transactions on Programming Languages and Systems 5(4), October, 1983.

[65] Sussman, Gerald Jay.
A Computer Model of Skill Acquisition.
American Elsevier Publishing Company, Inc., New York, 1975.

[66] Thayer, Thomas A., Liprow, M., and Nelson, Eldred C.
Software Reliability - A Study of Large Project Reality.
North-Holland Publishing Inc., New York, 1978.

[67] Tischler, R., Schaufler, R., and Payne, C.
Static analysis of programs as an aid to debugging.
SIGPLAN Notices 18(8), August, 1983.
(*Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on High-Level Debugging*, March, 1983).

[68] Weber, Janice Cynthia.
Interactive debugging of concurrent programs.
SIGPLAN Notices 18(8), August, 1983.
(*Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on High-Level Debugging*, March, 1983).

[69] Wirth, Niklaus.
Algorithms + Data Structures = Programs.
Prentice-Hall, Inc., New Jersey, 1976.

[70] Wirth, Niklaus.
Programming in Modula2.
Springer-Verlag, New York, 1982.

AUTHOR INDEX

Adam, Ann 8
Aho, V. Alfred 43
Amsterdam, Jonathan 42

Baiardi, F. 5
Balzer, R. M. 4
Bates, Peter 5
Beander, Bert 4, 5
Boehm, B. W. 17, 21

Cameron, Robert D. 32, 42, 55
Cardell, James R. 5
Cargill, Thomas A. 5
Cordell, Green 2

Dahl, O.-J. 2
Dannenberg, R. B. 1
DeMarco, Tom 2
Dijkstra, E. W. 22

Endres, Albert 17

Fairley, Richard E. 4, 11
Ferguson, Earl H. 4
Fosdick, D. Lloyd 6
Foxley, E. 32
Frenkel, Karen A. 2

Graham, S. L. 32
Grishman, Ralph 4
Gupta, N. K. 8

Hantler, Sidney L. 1
Hecht, Matthew S. 6, 26, 43
Hoare, C. A. R. 1, 22
Howard, William E. 1
Huang, J. C. 1, 32

Itoh, Daiju 4

Johnson, Mark Scott 5
Jones, T. C. 1

Kernighan, Brain W. 29

Lichtman, Zavdi L. 22

Manna, Zohar 2
Mauger, Claude 5
Muchnick, Steven S. 6, 26
Myers, Glenford J. 1, 3

Osterweil, Leon J. 6

Pai, Ajit B. 16
Popek, G. 22

Reynolds, J. C. 22
Richter, Helmet 16
Rubey, Raymond L. 17
Ruth, Gregory R. 8

Satterthwaite, E. 4
Sedlmeyer, Robert L. 7
Shapiro, Ehud Y. 7
Shooman, M. L. 1
Sippu, Seppo 16
Sussman, Gerald Jay 8

Thayer, Thomas A. 17
Tischler, R. 6

Weber, Janice Cynthia 5
Wirth, Niklaus 49, 76