# Error Directed Execution History Analysis: An Approach to Automatic Debugging

by

Edward Graham Okie

Dissertation submitted to the Faculty of the

Virginia Polytechnic Institute and State University

in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

in

Computer Science

APPROVED:

James D. Arthur, Chairman

Dennis G. Kafura

John W. Roach

Monte B. Boisen, Jr.

Ezra A. Brown

March 1989

Blacksburg, Virginia

**Error Directed Execution History Analysis: An Approach to Automatic Debugging**

by

Edward Graham Okie

James D. Arthur, Chairman

Computer Science

(ABSTRACT)

Execution history (EH) analysis is a major unexplored area in the development of debugging technology. In this dissertation we develop a theoretical foundation for incorporating EH analysis into the process of automatically debugging programs written in imperative, strongly typed, procedure oriented languages. This foundation includes the construction of a model for EH representation, an analysis of run time errors within the model, and the development of an approach to the use of EH analysis in automatic debugging. The model represents an execution history as a sequence of state vectors. Each vector contains both the values of program variables at a particular point in a computation and additional information that is used in the debugging process. Within this model, run time errors are classified by their effect on program termination, and characterized by their appearance within the EH. Based on this classification and characterization, techniques for detecting errors within an EH are presented. These techniques form the basis of an approach to automatic debugging in which a deterministic analysis locates errors in the execution history and, based on the results of this search, heuristic techniques perform automatic fault localization.

# Acknowledgements

I am very grateful to the many people who have contributed to the completion of this dissertation. In particular, I would like to thank Dr. James Arthur, my committee chairman. His ideas helped develop the initial focus of the project, his guidance kept me progressing on the right track, and his support and enthusiasm encouraged me to persevere to the end. I am truly grateful for the opportunity to have worked with and learned from him.

Also serving on my committee were Dr. Monte Boisen, Dr. Ezra Brown, Dr. Dennis Kafura, and Dr. John Roach. Their comments and questions provided valuable contributions to the early development of the research, and I am indebted to them for their fresh perspective on my work. Dr. Lenwood Heath also deserves thanks for serving, on very short notice, on the examining committee for my final defense.

From Dr. Donald Allison, Chairman of the Department of Computer Science, I received wise and valuable counsel concerning my graduate and professional careers. I am also indebted to him, and to , for their providing continual financial support during my long stay in the department. I also owe a debt of gratitude to the staff of the department. , , , and are among those who were always ready to help solve the day to day problems that I faced as a graduate student and teaching assistant.

It has been my distinct pleasure to have become friends with many of my fellow graduate students. In particular I think of , , , , , , , , , , , , and . These friends, and any others I may not have mentioned, have enriched my life, and I thank them all.

Of the many people outside the university who have contributed to the completion of this dissertation, one requires special mention. , my house group leader at the Dayspring Christian Community, has been a constant friend and I am indebted to him for the strong support and wise advice he has given frequently and freely.

I also owe much to my parents, , who provided me with a strong and rich family life, and who instilled in me the love of learning and the high standards that have made my academic life possible. We do not choose our parents, but if we did, I could not have made a better choice.

I have reserved until last my heartfelt thanks to my wife Sharon whose love, support, and affection have been a source of joy and strength amidst the rigors of completing this dissertation. I am truly blessed to have her as a wife, and it is to her that I gratefully dedicate this dissertation.

Finally, I wish to thank the Lord God for answering our prayers and enabling me to complete this dissertation. To Him be the glory.

# Table of Contents

# List of Illustrations

# 1.0 Introduction

Interactive debuggers are highly valued program development tools [HANS85]. This fact, combined with the high cost of program development and maintenance, has motivated improvement in debugging technology in the areas of multilanguage[BEAB83], visual [ISOS87], and intelligent debuggers [SEVR87]. Despite this progress, however significant gaps in debugger research continue to exist. One deficiency is the neglected use of execution history (EH) concepts as a formal basis for characterizing the debugging process. Plattner refers to EH analysis as a major unexplored research area [PLAB81]. Although McCarthy and others [MCCJ66, LANP66] use a state vector model to define program semantics, this work unfortunately is not directed toward the analysis and detection of program errors. Work that does use EH analysis as a debugging tool [BALR69, FAIR75] focuses on developing specific, isolated techniques rather than creating a coherent theory of EH-based error analysis and debugging. Specifically, there are no studies that address the appearance of errors in a program's dynamic behavior and that relate the dynamic characteristics of errors to the debugging process.

In reaction to this deficiency the research described in this dissertation lays the theoretical foundation for the development of an intelligent debugger that (a) performs *automatic* execution history analysis based on existing error conditions, and (b) provides smart commands which support user-directed error analysis as a supplement to the automatic operations. The function of the automatic analysis is to examine a program and an instance of that program's execution history and to provide, where possible, information concerning (1) errors that appear within the execution history, (2) source code conditions underlying these errors, and (3) additional distilled EH and source code characteristics that may be relevant to the debugging process. If automatic analysis does not provide sufficient information, then with smart commands the user can direct an EH analysis that provides additional information in support of manual debugging. Underlying the development of this system is a research effort that focuses on the four major areas discussed below.

The first area is the development of a model for representing a program's dynamic behavior as an execution history. The model will focus on the representation of imperative, procedure oriented, strongly typed languages. Within the model, an EH should represent the actual behavior of a program executing in a single processor environment, including the behavior of individual statements and the flow of control among statements.

Based on this model, the second area of research is the classification and characterization of run time errors with respect to their appearance within an execution history. Similarly, because the debugger is assumed to operate with no explicit specification of the desired behavior of the source program, errors are restricted to those actions that violate *general* standards of correct program behavior. Specifically, production of incorrect output is not considered to be an error (although smart commands can be used to explore causes of such errors).

The third research area is devising an approach to automatic debugging that uses the error classification and characterization scheme described above. This approach should define methods for using the EH characteristics to locate errors within the EH, to determine source code statements that can underlie these errors, and to distill from the source code and the EH certain characteristics that may be relevant to the debugging process. These characteristics may indicate specific source code statements that appear to underlie the error or they may only provide information that is likely to be related.

The final area is the construction of a prototype debugger that implements the approach defined in the third research area. In developing this prototype, emphasis should be given to solving practical problems associated with EH collection and representation and to devising and implementing the automatic debugging techniques. To simplify the design and to maintain the research focus, program and error information as well as automatic debugging techniques will initially be hard-coded. Based on the results of constructing and using this prototype, problems such as representing error characteristics and detection techniques, and developing a graphics based user interface can then be addressed.

Of the four research areas just outlined, the first three emphasize the theoretical aspects of debugger development, while the final area provides practical testing of the theory. This dissertation describes research in the first three areas and thus provides a theoretical foundation for EH based debugging. Major results of the research effort include

- a model in which a program's execution history is represented as a sequence of state vectors,
  - a definition of the elements of the state vector, and

- a definition of transition functions that relate statement execution to EH vector modifications,
- a scheme for classifying execution histories, based on their termination status,
- a scheme for classifying errors based on their effect on program termination,
- a characterization of the occurrence of individual errors within a state vector sequence execution history, and
- algorithms that implement an approach to automatic debugging which is based on the relative importance of errors and their occurrence within a specific execution history.

This disseration describes these results in detail.

Chapter 2 provides background for the research, including a discussion of the evolution of debugger technology and of current research in creating intelligent debuggers. It also describes the languages that are considered in the research and the processing environment that is assumed to produce an EH as it executes programs written in these languages.

Chapters 3 and 4 define the model for EH representation. Chapter 3 describes the state vector characteristics that allow the vector to represent a snapshot of a program at a particular point in a computation. Chapter 4 defines (a) the functions that model the state vector to state vector transitions which represent the execution of individual program statements and (b) the state vector sequences that serve as a history of program execution. Chapter 4 concludes with a discussion of a scheme for classifying execution histories.

Chapter 5 discusses errors. After defining the term error, it presents an error classification scheme that is based on an error's effect on program termination, and defines distinguishing characteristics of individual errors. This chapter also considers error detection and analysis techniques that are derived from the error characteristics; it concludes with a discussion of the relationship between error and execution history classes.

Based on the EH analysis of Chapters 3 and 4 and the error analysis of Chapter 5, Chapter 6 defines a particular automatic debugging strategy and defines the concepts and algorithms needed for its implementation. The final chapter summarizes the results presented in the dissertation and describes directions for future research.

# 2.0 Background

To provide background for the dissertation research, this chapter describes the development of debugging technology and discusses current research aimed at creating intelligent debuggers. In addition, it outlines assumptions concerning the class of languages considered in this research and the processing environment for program execution.

## *2.1 Development of Debugging Technologies*

To place this research in perspective, this section outlines the development of debugging technologies with respect to the types of information used during the debugging process. This information falls into two categories:

1. static information that is derived from the source program, and
2. dynamic information that is available in the execution history.

The following examination of debugger development reveals that as debuggers have matured, they have made use of progressively larger subsets of this static and dynamic information. Typical subsets of static information include the program's symbol table and the text of the current source statement being executed; earlier debuggers exclude information deriveable from the program's logical structure and data flow. Similarly, initial subsets of the dynamic information include dumps of the current (or last) program state and/or a trace of machine language statements executed; a complete execution history is used by some later debuggers. The above observations reflect the partitioning guideline of the following discussion on debugger evolution.

### 2.1.1 Debuggers with trace and dump facilities

Among the earlier attempts to support the debugging process, one finds the special hardware and machine language facilities of the EDSAC [WILM57]. In addition to being one of the first stored-program computers, the EDSAC supported debugging facilities such as the setting of breakpoints, the dumping of memory when breakpoints are encountered, and providing an instruction trace of program execution. These features, later indigenous to many computers, are seen for the first time on this system. Expanding on the capabilities of the EDSAC, the EDSAC-2 incorporates trace collection routines that also condense the trace of large loops [BARD63]. A tendency to further exploit execution characteristics becomes more apparent in later debuggers, and is a focal point of this research.

As typified by the EDSAC series, early debugging facilities utilize only a portion of the available execution history information and no source code information. Consequently, corresponding memory dumps and breakpoint specifications are numerically oriented. The class of debuggers discussed next utilizes source code information in the debugging process, and hence, significantly reduces the numerical orientation.

### 2.1.2 Debuggers that use partial source code information

Around 1960 *symbolic* debuggers appear, e.g. the debugger for the IBM 709 SHARE system [GREI59] and the interactive debugger DDT [KOTA61]. In addition to execution histories, these debuggers also exploit source code characteristics by presenting symbolic dumps in source code terms and by allowing break points to be specified relative to source code statements. Based on the effectiveness of symbolic interaction demonstrated by these systems, additional high-level language debuggers emerged. Resulting systems include MADBUG [FABR65], Quiktran [DUNT64], the PL/1 Checkout compiler [CUFR72] and Algol-W [SATE79]. Typically, these types of debuggers allow the user to query and display information from a high-level, source code perspective. Although this approach simplifies debugging, it creates new implementation problems, e.g. mapping multiple machine language statements to a single source language statement and the handling of variable scope [HOLD83]. Moreover, these debuggers force multi-language programmers to learn the operational characteristics of multiple debuggers because each debugger is affiliated with only one high-level language (as implied by its name). Later developments, incorporated in debuggers like the UNIX sdb [AT&T84] and VAX DEBUG [BEAB83], resolve these problems and provide a single debugging environment for multiple languages.

The debugger systems discussed thus far use only part of the available information base. In particular, the synthesis of information based on static data flow analysis and dynamic runtime characteristics are ignored. Debuggers of the next two subsections provide this synthesis.

### 2.1.3 Debuggers that use the entire execution history

In a survey on monitoring program behavior [PLAB81], Plattner describes execution history collection and analysis as a major unexplored research area. Initial approaches to collecting execution histories are hardware based and focus on tracing machine instructions [MANS84]. A history gathered at such a fine level of granularity, however, is of limited use because of the data volume. A more practical approach, exemplified by EXDAMS [BALR69] and ISMS [FAIR75], is to collect an execution history at the source statement level. Using this approach, these systems provide a collection of debugging techniques that are based on EH analysis, including forward and reverse execution of a program, determination of variable value ranges, and flowback analysis. Flowback analysis [BALR69] searches backwards through an EH to find the statements that have contributed to the value of a given variable at a given statement.

Execution history based techniques for debugging parallel programs are available in the system PPD [MILB88]. This system limits the amount of data in the EH by collecting information only at certain required points, as determined by a static analysis of the program. Data for points between these required snapshots is calculated as necessary by re-executing the program from the snapshot point. Analysis of the EH thus collected allows the system to create dynamic program dependency graphs that can be used to detect race conditions and other synchronization errors. Although this debugger has the capability to detect certain errors common to parallel programs, no attempt is made to detect the fault underlying such errors or to use this approach to create an intelligent debugger. One debugger that does use execution analysis as a basis for intelligent debuggers is MTA - the Message Trace Analyzer. This system is discussed in section 2.2 below.

### 2.1.4 Debuggers that use information from the full source code

Data flow analysis, first an aid to global code optimization [ALLF74, ALLF76] and later a technique for source code anomaly detection [FOSL76, OSTL81], is the foundation for debuggers such as DAVE [OSTL76], FAST [BROJ78], and SADAT [VOGU80]. These debuggers analyze a program's logical structure to find data flow errors including references to undefined variables and multiple variable definitions without an intervening reference. Building on data flow analysis, another debugging technique, called *slicing* [WEIM82, WEIM84], provides a list of the program statements whose execution affects the value of a given variable at a given statement. In debugging with such a technique, a programmer slices using a variable whose value is incorrect and then examines the statements in the slice to find the source of the error. The technique *dicing* reduces the size of this list by eliminating statements that also occur in the slice of a variable whose value is known to be correct. Within the assumptions outlined in [LYLJ84], slicing and dicing provide limited automatic fault localization. Other debugging systems that use program analysis to provide automatic fault localization are described in the following section.

## 2.2 *Intelligent Debugging Systems*

To support the debugging process, some current debuggers utilize *both* the program's source code and execution history. Rather than focusing on new debugging techniques, however, current research is being directed toward making debuggers more intelligent, i.e., giving them the ability to detect and correct errors.

Because debugging is such a difficult task, there is a critical need for "intelligent debuggers" that actively assist the programmer in detecting errors and locating their sources. The debugging systems presented below are considered to be more "intelligent" than those described in the previous section because they

1. are aware of fundamental programming principles, and/or
2. understand specific properties of the program being debugged.

This knowledge, along with the source code and execution histories, allows current debuggers to detect a larger class of errors. Woodpecker [FOUJ84], for example, uses its knowledge of language structures and programming techniques to address issues such as the interchangeability of statements and redundant use of variables. It also has the capability to modify the program to effect error correction.

The system PHENARETE [WERH82] uses its knowledge of LISP syntax and functions to first transform the program it receives as input into a syntactically correct and semantically consistent program. Symbolic execution then produces a program representation that can be checked against PHENERETE's general rules about correct programs. If a discrepancy is found, the system attempts to correct the error by using rules that describe program repair. For example, PHENARETE flags a recursive function that lacks a base case and repairs the function by adding the code it thinks is missing.

A third debugging system, PROUST [JOHW85], requires a description of the user's program in the form of a *plan*. Associated with elements of the plan are descriptive implementation scenarios based on the underlying programming language. PROUST tries to match the user's program with possible implementations of the program plan and then locates errors by detecting discrepancies between the plan implementation and the user's program.

Similarly, PUDSY [LUKF80] requires that the programmer provide assertions that describe various sections of the code. By finding discrepancies between the user provided assertions and the assertions that it derives from the code, it is able to detect and possibly correct sections of the code that do not perform as expected.

Shapiro [SHAE82], on the other hand, uses the plan approach but takes a different approach to representing the user's program. The user specifies the functionality of the program and its constituent functions by giving expected output for given input. Shapiro's system then looks for errors by comparing the expected output with the actual output. It also analyzes the history of function calls to detect possible causes of an output discrepancy. Underlying Shapiro's system is a description of errors based on execution history characteristics. This approach is similar to that described in this dissertation, but Shapiro uses a functional program history in a declarative language (Prolog), rather than a state sequence program history in an imperative language. Furthermore, our system does not require *any* representation of the program's intended behavior; error analysis is based exclusively on relationships among source code and execution history characteristics, and on general assumptions concerning correct program behavior.

The execution history analysis approach is also used by the system MTA, the Message Trace Analyzer [GUPN84]. This system compares a finite state machine representation of a system's intended behavior with a history of interprocess messages created by the system. Any discrepancy between the machine specification and the system's actual behavior indicates a fault in the system. Thus this system uses a type of plan to guide its analysis of an execution history.

This discussion of intelligent debuggers complete the description of debugging development and research. The following two sections complete the background discussion with a description of the languages that are considered and the processing environment that executes programs from these languages.

## 2.3 Language Levels

Our model is designed to represent the execution behavior of programs written in imperative, procedure oriented, block structured languages. It can handle static variables of simple, array or record types as well as dynamically created variables of simple or record type. It also accommodates the following fundamental control structures: sequence, if_then_else and while loops. Programs are assumed to be strongly typed, to compile correctly and to have no self-modifying code. To simplify the model, we exclude input and output from consideration, leaving them as an obvious area of future work. For effective communication, examples use the syntax of Pascal. More complete details of the language are given in Chapter Four.

Based on the features outlined above, we partition programming languages into the hierarchy shown in Figure 1 on page 14. Level 1, the simplest, exemplifies lan-

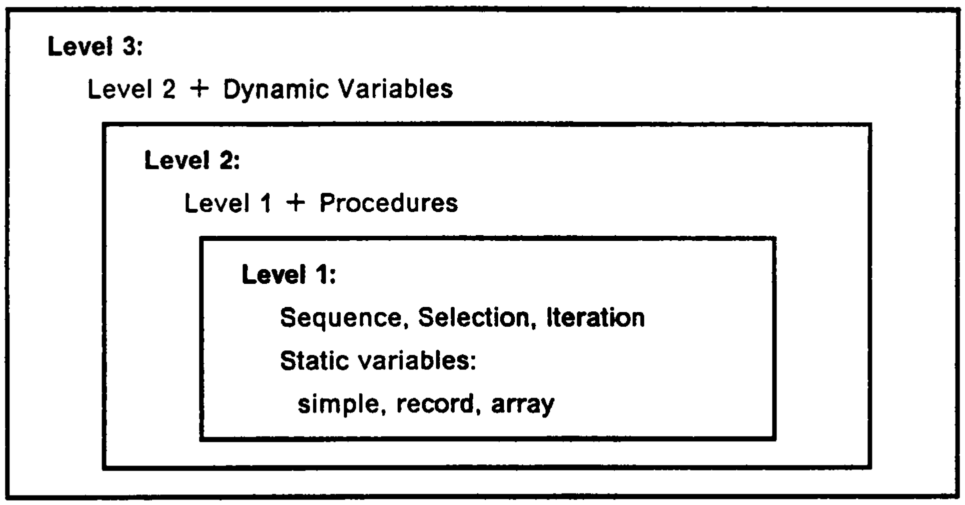


Figure 1. The Language Hierarchy

guages that have the three basic control structures and static variables, but not procedures or dynamic variables. The second level extends the first by including languages that have procedures; and the highest level includes the lower two levels and extends them by incorporating languages that allow dynamic variables. Using this hierarchy, the next two chapters describe a hierarchical model for representing the execution of programs written in languages from each of the three levels. The model for Level 1 languages is the simplest. Additions to that model make it capable of describing the execution history of Level 2 and Level 3 languages. This division allows the description of simple model and language features before more complex ones are introduced.

## *2.4 The Processing Environment*

We assume the existence of a processing environment (PE) that takes a source program as input and that produces an execution history which reflects execution of the program. We assume that the environment executes the program sequentially with a single processor that has finite storage capacity. Further assumptions about the environment determine the conditions that cause a computation to abnormally terminate. For example, an array reference whose index exceeds the array's declared bounds may or may not cause the computation to terminate immediately and abnormally. (We assume that it does.) The exact conditions that cause abnormal termination are outlined in chapter five. We also make assumptions about the information that is available from the PE, without considering how this information is computed. The information that must be available includes:

- the termination status of the computation, either
  - normally terminated,
  - abnormally terminated, or
  - not terminated;
- the addresses of
  - global variables,
  - procedure parameters and local variables, and
  - dynamically allocated variables; and
- the initial values of all memory elements.

The termination status is not part of the execution history model of computation, but it is used in the debugging process. Specifically, an abnormally terminated termination status is the indication that an error has occurred and the computation is unable to continue. The latter two items of information, addresses and initial values, allow the model to accurately represent (a) the effects of using undefined variables and (b) wild addressing (that is, illegally accessing a memory location by referencing its address.) Obtaining this information directly from the PE allows the model to be accurate but not overly complex.

With this background, the next two chapters describe the model for EH representation. Succeeding chapters describe errors and automatic debugging.

# 3.0 An Execution History Model

## 3.1 Introduction

The first of the three major contributions of this research effort is an appropriate computational model for effective execution history based debugging. Tailored to the representation and analysis of a program's dynamic behavior, including errors that occur in a program's execution, our model represents a computation as the sequence of states. Each state is a snapshot of the program at an instant in the computation, and the transition from one state to its successor represents the execution of a designated program statement. Following McCarthy [MCCJ63], we represent a state as a state vector that contains the values assigned to each of the program's variables, and we define *state transition functions* to describe the state to state transitions caused by the execution of program statements. These functions are defined in the next chapter; sections 3.4, 3.5 and 3.6 of this chapter define the structure and properties of the state vectors, but first, section 3.2 examines other models of computation and section 3.3 presents an overview of the model.

## 3.2 Background

In order to precisely define techniques for execution history based debugging, an appropriate model of computation is essential. We consider a model's representation of the history of a program's execution to be appropriate if it:

- Describes the behavior of a program in an actual processing environment, including
  - the action of individual statements and
  - the flow of control among statements, and
- Is suitable for the development and application of effective debugging techniques.

To meet these requirements we use a model of computation similar to those used to define an operational semantics of a programming language. Before discussing operational models we examine other, inadequate, computational models relative to these requirements.

In models of computation that are used to explore the meaning and limits of computability, computations bear little resemblance to the execution of programs written in high level languages. Consequently, models such as Recursive Function Theory [ROGH67], the Lambda Calculus [CHUA41] and Turing Machines [TURA36], are inappropriate.

Models of computation underlying axiomatic [FLOR67, HOAC69] and denotational semantics [TENR76] also prove inadequate. Geared toward program verification, the former model associates axioms and rules of inference with statements to specify the effect of a program's execution. Unfortunately the mechanism of achieving the effect is not specified, and consequently there is no detailed history of the program's behavior. For example, one effect of the following code

```
P: if X > Y then SWAP (X,Y);
Q:                      {Assertion: X ≤ Y}
```

is the establishment of { Assertion $X \leq Y$} at point Q, but this assertion gives no indication of whether the then part of statement P has actually been executed, an essential part of an execution history.

Although an axiomatic model is not suited to capturing an execution history, an axiomatic approach to program debugging is mentioned (first) by McCarthy [MCCJ63] and further developed for example by Lukey [LUKF80]. The positive results of that work suggest the benefits of a synthesis of the axiomatic and execution history approaches to debugging, similar to complementary definitions of program semantics [DONJ76, LAUP71]. Because it is beyond the scope of this dissertation, such a synthesis is not further considered.

In contrast to the axiomatic model, the model of computation underlying denotational semantics contains the information necessary to represent an execution history; however, this information is not easily available, and consequently the model is not suitable for developing debugging techniques. In this model, the meaning of a program is defined by a function that maps an initial state to a final state. This function defines the overall behavior of a program, but not the behavior of individual program statements, as needed for an execution history. This statement level behavior is implicitly represented in the intermediate states that are passed between the intermediate functions of which the meaning function is composed, but as the model is designed, it does not represent this information explicitly.

The best approach to capturing the history of a computation is an operational one in which the sequence of states that is assumed by an abstract machine both represents the computation and serves as its execution history. A state accurately models the processing environment's memory component, and changes to the state closely follow the actual effects of statement execution. Furthermore, using a particular state vector element as a pointer to the next statement in the execution sequence provides a record of control flow. Finally, the error analysis described in chapters four and five of this dissertation demonstrates the suitability of this method for the development of debugging techniques.

The operational approach can be traced to the work of McCarthy [MCCJ63, MCCJ66] and Elgot [ELGC64]. McCarthy's earlier paper details a mathematical basis for computer science in which recursive functions on state vectors are used to prove properties of programs, and in doing so it introduces the concepts of (a) a state that holds the values assigned to variables and (b) using state to state transition functions to define the meaning of statements. In the earlier paper, McCarthy models flow of control with mutually recursive functions in a method similar to the approach of denotational semantics; in the later paper he uses a state vector element that points to the statement that is about to be executed. We include this modification in our model because it more closely portrays the action of an actual processor. Elgot introduces the notion of modeling a computation as a sequence of states assumed by an abstract machine. Although he proposes, without elaboration, using his abstract RASP machine to define programming language semantics, his states and machine operations do not model high-level language concepts.

Building on this early work, two distinct variations of the operational approach have been developed: Landin's SECD Machine [LANP64] and the Vienna Definition Language (VDL) [WEGP72]. Unfortunately, neither variation provides an execution history that is suitable for developing debugging techniques. Landin's approach is to translate the source program into abstract expressions (generalizations of lambda expressions) and to interpret these abstract expressions with an abstract machine that is specially defined for the purpose, the SECD (State, Environment, Control, Dump) Machine. This machine provides a workable specification of a high level language; but its choice of machine language, abstract expressions, forces a computation to be defined by expression application and evaluation, rather than the imperative commands of an actual processor. Consequently the execution history produced by such a machine does not portray a program's behavior closely enough to develop effective debugging techniques.

The Vienna Definition Language, the most successful formulation of operational semantics, is powerful enough to specify both the context sensitive syntax and the semantics of sophisticated languages such as PL/1 [LUCP69], BASIC [LEEJ72] and (a subset of) SNOBOL [PAGF78]. However, because we are concerned only with modeling the execution behavior of programs and not with giving a full semantic definition of a language, we do not need the definitional power provided by the VDL and hence, to avoid the complexity underlying its power, we do not use its underlying abstract machine.

Two illustrations of this unnecessary complexity follow. First, the control substate is a tree whose branches represent execution alternatives where execution order is not fixed, as in PL/1 expressions. Because the languages we model are assumed to be completely deterministic, this capability is not needed. Second, both static program structures as well as runtime data structures can be stored in the state and operated on by machine instructions. This dual nature of the instructions and the state elements, an important innovation of the VDL, unfortunately makes the sequence of states assumed by the machine unsuitable as a history of execution of a program. In such a sequence, many of the state to state transitions reflect manipulations of the program rather than reflecting actions caused by executing program statements. For example, the state transition caused by executing a statement list does not show the first statement being executed; instead it shows the statement list being split so that the first instruction is ready to be executed. Existing operational models, therefore, do not provide a suitable execution history. An overview of a model tailored to this task is the subject of the next section.

## 3.3 Overview of the Model

In this section we describe the main features of the model we have developed for execution history representation. The essence of an operational model is an abstract machine that assumes discrete states during a computation. Such a machine is defined by (a) the contents and structure of its states and (b) the functions that cause transitions between states. In defining our model we therefore concentrate on describing the state and the state to state transition functions. The structure of a state depends on the characteristics of the language that is being modeled. Therefore, this overview describes the state characteristics that are required by the language levels described in the previous chapter. It also provides an illustration of the execution history produced by an example program and, within the context of that illustration, describes example state to state transition functions.

The execution behavior of a program that contains only static variables and neither user defined procedures nor pointer semantics can be modeled by a sequence of state vectors where each vector is a collection of elements, and each vector element corresponds to a single simple variable or complex variable component. To model flow of control in such programs, each state vector includes an element whose value represents the sequence number of the next statement in the execution sequence. Such a model represents the features of Level 1 Languages; complex control structures and data types require a more complex model. Allowing state vector elements to assume stack characteristics permits the modeling of procedures, while representing dynamically allocated variables requires that the state vector grow and shrink to accommodate variables that are created and destroyed. Including dynamically allocated variables in the model introduces the possibility of wild address errors, which occur if pointer variable accesses another variable of the wrong type. Modeling these errors requires that each state vector element contain the physical address of the memory cell allocated to the variable modeled by that element. With the additional state characteristics of a varying size and of addresses the model can represent the languages described above.

One further addition enhances the debugging process: each state vector element is augmented with information that describes the source of its value. Each element then is a 4-tuple (V, P, S, A) where

- V is the value attributed to the variable associated with the vector element,
- P is the sequence number of the program statement that produced the value V,
- S is the sequence number corresponding to the state vector at which statement P was executed to produce value V, and
- A is the address associated with the vector element.

This extension enables, for example, the identification of assignment statements that do not change the target variable's value and the detection of wild address operations.

To get a better grasp of the state vector approach to representing execution history, consider the example program and history shown in Figure 2 on page 25. For simplicity, assume a Level 1 language, i.e. one with no procedures or dynamically created variables. Because these languages require neither stacks nor addresses, they are modeled by a fixed length state vector whose elements are 3-tuples.

In this example, $S^i$ represents state vector i of the execution history sequence. Each vector contains elements for the three variables A, B and C and an element NS (Next Statement). NS is used throughout this dissertation to represent the number of the state vector element that specifies the number of the statement that is next in the execution sequence. Each element of the state vector is a 3-tuple denoting, in order, the value of the variable, and the numbers of the statement and the state at which that value was assigned. Every statement, or test part of a conditional statement, (i.e. if or while test), has four attributes, NUMBER, TD, FD, and NEXT. The first attribute, NUMBER, provides unique sequence number; NEXT identifies the logical successor of non-conditional statements; and for conditional statements the true and false destination attributes (TD and FD) respectively specify the statement to execute next if the condition is true or false respectively. Assigning the value of the NEXT, TD or FD attribute to state vector element NS in successive states models the flow of control from one statement to the next.

As initial conditions, this program has the values shown in state vector $S^0$, in which NS indicates the statement that is to be executed first and the remaining variables have system defined initial values. The transition from state 0 to state 1 shows

| ATTRIBUTES | | | | SOURCE PROGRAM | STATE VECTOR EXECUTION HISTORY | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | | | | i | $S^i$ | | | |
| NUMBER | NEXT | TD | FD | | | NS | A | B | C |
| | | | | Program Example1; | | | | | |
| | | | | Var A, B, C: Integer; | | | | | |
| | | | | Begin | | | | | |
| | | | | | 0 | 1,U,0 | $I_1$,U,0 | $I_2$,U,0 | $I_3$,U,0 |
| 1 | 2 | - | - | A := 10; | | | | | |
| | | | | | 1 | 2,1,1 | 10,1,1 | ″ | ″ |
| 2 | 3 | - | - | B := 12; | | | | | |
| | | | | | 2 | 3,2,2 | ″ | 12,2,2 | ″ |
| 3 | - | 4 | 5 | If A > B then | | | | | |
| | | | | | 3 | 5,3,3 | ″ | ″ | ″ |
| 4 | 6 | - | - | C := 20; | | | | | |
| | | | | Else | | | | | |
| 5 | 6 | - | - | C := 30; | | | | | |
| | | | | | 4 | 6,5,4 | ″ | ″ | 30,5,4 |
| 6 | U | - | - | A := A + C | | | | | |
| | | | | | 5 | U,6,5 | 40,6,5 | ″ | ″ |
| | | | | End. | | | | | |

- U denotes UNDEFINED.
- - indicates that the statement has no value for the attribute.
- $I_1$, $I_2$, $I_3$ are the system defined initial values of the elements.
- ″ indicates no change in the vector element.
- Each state vector element is a triple (V, S, N) where
  - V is the content value,
  - S is the number of the statement that assigns value V, and
  - N is the number of the state that follows the value change.

**Figure 2. An Example Execution History**

the two actions of the assignment statement as it replaces the values of elements A and NS. This transition is modeled by the function

$$\lambda S.\text{REPLACE}(S, NS, (2,1,1)) \circ \lambda S.\text{REPLACE}(S, [A],(10,1,1))$$

where REPLACE(S, e, t) replaces the current triple of element e of state vector S with the triple t. The notation [V] denotes the element of S that corresponds to variable V . In Chapter 4 we define the utility functions needed to map variables to elements, evaluate the expression on the right hand side of an assignment and select the NEXT attribute of a statement.

The transition from state 2 to state 3 models the change of control flow caused by the if statement. It is modeled by

$$\lambda S.\ \text{if } (A > B) \text{ then REPLACE}(S, NS, (4,3,3)) \text{ else REPLACE}(S, NS, (5,3,3))$$

which causes NS to be replaced with the false destination, 5, because the condition, (A > B), evaluates to false. If the condition had been true then statement four would have been executed next, followed by statement six, as specified by statement four's NEXT attribute. States four and five model assignment statements five and six, and the computation terminates when NS is replaced by statement six's NEXT attribute: UNDEFINED.

Finally, note that element NS of each state vector (except the last) of an EH specifies the unique statement that is to be executed to create the next state vector in the state vector sequence. Thus we can unambiguously refer to the statement that *follows* any state (except the last) or the statement that *precedes* any state (except the first). Therefore, in the sample EH above, we say, for example, that statement 1 follows state 0 and precedes state 1.

## 3.4 A Model for Level 1 Languages

As mentioned earlier, the definition of the model requires the definition of (a) the state vectors that model the system state and (b) the state transition functions that model execution of program statements. This section describes the structure and elements of the vectors needed to model Level 1 languages. Also described are the *action* functions that are used, in Chapter 4, to define the state transition functions.

### 3.4.1 State Vectors

The state vector captures the static condition of a program at a particular instant of the program's execution. At Level 1, each element of a vector is a 3-tuple consisting of the value currently associated with a specified program variable plus a statement and state number that describe the origin of that value. State vectors are members of the set STATES; vector elements, of CELLS; and element values, of VALUES.

### *3.4.1.1 VALUES*

Part of the definition of any programming language is the specification of a set of atomic values that can be assigned to simple variables or to simple components of structured variables. For Level 1 languages the set contains at least the integers in the range [MININT..MAXINT], some set of real numbers, the usual character set, and the boolean values. Without specifying this set further, we define

VALUES = { v | v is an assignable value }.

### *3.4.1.2 CELLS*

To enhance debugging analysis, each state vector element is expanded to a 3-tuple consisting of a content value, a source code statement number and a unique execution history state designator. In essence, each triple conveys a content value of its associated variable, the source code statement that effected that content value, and an execution history time stamp reflecting when the effect occurred. We therefore describe the set of elements as

CELLS = $\{ (v,i,j) \mid v \in$ VALUES, $i,j \in$ Integer$\}$.

We call an individual member of CELLS a cell and use the following notation to select components of a cell:

For $t = (a,b,c) \in$ CELLS, $t|_1 = a$, $t|_2 = b$ and $t|_3 = c$.

Modeling Level 3 languages requires CELLS to be 4-tuples. Thus we extend the notation to select a component of an arbitrary n-tuple.

The additional information associated with each variable is essential for effectively tracing the statements that influence an element of a computation. One debugging technique, for example, verifies that the value of each variable in an erroneous statement is assigned legally and not by a wild address operation. Other techniques are described in Chapter 6.

### *3.4.1.3 STATES*

An element of the set STATES is a vector whose elements are 3-tuples (i.e. cells) and whose length is fixed by the program being modeled. Where n is the vector length, we define

$$\text{STATES} = \{S \mid S = \langle s_1, s_2, \ldots s_n \rangle, \text{ where } s_i \in \text{CELLS}, \text{ for } 1 \leq i \leq n\}$$

Extending the notation of the previous section, if

$$S = \langle s_1, s_2, \ldots s_n \rangle$$

then $S_i$ denotes $s_i$, the ith element of S; and $S_i|_j = (S_i)|_j$ selects the jth member of the triple $s_i$.

## 3.4.2 Action Functions

Our goal is to define state to state transitions that model the execution of statements from a Level 1 language. Each such transition involves the replacement of one or more state vector elements with new 3-tuples. We therefore define an action function REPLACE that replaces a single, designated element, and we use this function to define state transition functions that model program statements. Formally, REPLACE is defined as follows.

For $S = <s_1, ..., s_n>$,

REPLACE: STATES $\times$ INTEGER $\times$ CELLS $\rightarrow$ STATES: $(S, i, c) \mapsto <t_1, ..., t_n>$

where $t_j = s_j$, for $1 \leq j \leq n, j \neq i$, and

$t_i = c$.

The example of section 3.3 shows the use of REPLACE in the state transition function used to model specific assignments and if statements. The functions that model general assignments, if, and while statements are shown in chapter four.

## 3.5 *A Model for Level 2 Languages*

### 3.5.1 The State Vector

Extending Level 1 languages by adding procedures produces the class of Level 2 languages. Modeling these languages requires that the state vector be modified so that its elements take on stack characteristics. To manipulate these stacks we introduce primitive stack functions and use them to define the action functions that perform stack operations on selected state vector elements. In this section we define the set of state vectors and the set of stacks that are their elements. The stacks are formed of triples that are taken from the set CELLS. The sets VALUES and CELLS are defined as in Level 1 and are not redefined here.

#### *3.5.1.1 The domain STACKS*

The set of stacks that are used as state vector elements is defined simply as

STACKS = { stk | stk is a stack whose members are in CELLS }.

The following functions operate on the individual members of the set STACKS. These assumptions:

s is an arbitrary member of STACKS and

c is an arbitrary member of CELLS,

are used in defining the stack functions.

- newstk: $\rightarrow$ STACKS

  newstk = an empty stack
- addtop: STACKS $\times$ CELLS $\rightarrow$ STACKS

  addtop(s, c) = s with c pushed onto it
- deltop: STACKS $\rightarrow$ STACKS

  deltop(s) = s with its top cell removed
- top: STACKS $\rightarrow$ CELLS

  top(s) = the cell that is on top of s
- isempty: STACKS $\rightarrow$ BOOLEAN

  isempty(s) = (s $\equiv$ newstk)

There are no surprises in these function's operations; they follow the usual axioms:

deltop(newstk) = ERROR

deltop(addtop(s, v)) = s

top(newstk) = ERROR

top(addtop(s, v)) = v

isempty(newstk) = true

isempty(addtop(s, v) = false

### *3.5.1.2 STATES*

Except for having members of STACKS and not CELLS as individual elements, the set of STATES is defined as in Level 1. For some $n \geq 1$ (where n is determined by the program that is being modeled) we define

STATES = $\{ S \mid S = <s_1, s_2, ..., s_n>$, where $s_i \in$ STACKS, $1 \le i \le n \}$.

Following the notation from Level 1, top($S_i$) denotes the top cell of element i of state vector S, and the members of this triple are denoted by

top($S_i$)$|_1$, top($S_i$)$|_2$ and top($S_i$)$|_3$, respectively.

We have introduced an ambiguity here - the name STATES refers to different sets in the Level 1 and Level 2 models. If it is not clear from the context which set a reference to STATES specifies, then we will explicitly make the distinction.

Because of the inclusion of stacks in the model, new action functions are needed to perform the stack operations on designated elements of the state vector. These functions are the subject of the next section.

### 3.5.2 Action Functions

In modeling Level 2 statements, the operations we may wish to perform on an individual element of a state vector are to push and pop triples onto or off of a designated stack. These operations are performed by the action functions PUSH and POP. No program statement can form a new stack, and hence we do not define a NEWSTACK action function. In addition to the stack functions, a new version of the action function REPLACE is introduced for the newly defined state vectors. For each of the definitions below,

S $\in$ STATES is defined as S = $<s_1, ..., s_n>$ where $s_i \in$ STACKS, for $1 \le i \le n$.

The definition of the two stack modifying action functions follows.

- PUSH modifies a state vector by placing a triple onto a specified element (i.e. stack) of the state vector.

PUSH: STATES × INTEGER × CELLS → STATES: $(S, i, c) \mapsto \langle t_1, \ldots, t_n \rangle$,

where $t_j = s_j$ for $1 \leq j \leq n, j \neq i$

$t_i$ = addtop($s_i$, c).

- POP modifies a state vector by removing the top member from a specified element.

POP: STATES × INTEGER → STATES: $(S, i) \mapsto \langle t_1, \ldots, t_n \rangle$,

where $t_j = s_j$ for $1 \leq j \leq n, j \neq i$

$t_i$ = deltop($s_i$).

The newly defined STATES require a new function REPLACE′ to replace the individual cell that is the current value of an element's associated variable.

- REPLACE′ modifies the state vector by replacing the triple that is currently on the top of a specified stack. It is similar to the Level 1 REPLACE except that REPLACE operates on elements that are single cells and REPLACE′ operates on elements that are stacks of cells. Formally,

REPLACE′: STATES × INTEGER × CELLS → STATES: $(S, i, c) \mapsto \langle t_1, \ldots, t_n \rangle$,

where $t_j = s_j$ for $1 \leq j \leq n, j \neq i$

$t_i$ = addtop( deltop($s_i$), c).

The stack functions PUSH and POP allow modification of the top of a stack and consequently the action of REPLACE′ can be modeled by a PUSH and a POP. Specifically,

$$\text{REPLACE}'(S, i, c) \equiv \text{PUSH}(\text{POP}(S, i), i, c).$$

### 3.5.3 An Example State Transition Function

To illustrate the new Level 2 action functions, Figure 3 on page 36 shows the state transition function that models the calling of a subroutine. In this example, assume that:

- Statement 11 is the first executable statement in the subroutine,
- State 55 precedes the call to the procedure,
- Statement 20's successor is statement 21 (not shown), and
- Elements A, B, and L are empty stacks before P is invoked.

The call being modeled causes the following actions:

- The initial value of the local variable is pushed onto its corresponding element,
- the value of each value parameter is pushed onto its respective element,
- the return address is stacked onto element NS, and
- the entry point of the called procedure is stacked onto element NS.

These actions are reflected in the transition from state vector 55 to state vector 56 and are modeled by the function

$\lambda$S.PUSH(S, NS, (11,20,56)) o $\lambda$S.PUSH(S, NS, (21, 20, 56))

o $\lambda$S.PUSH(S, vpindex(S, 2), (31,20,56)) o $\lambda$S.PUSH(S, vpindex(S, 1), (30,20,56))

o $\lambda$S.PUSH(S, lvindex(S, 1), (ivalue, UNDEFINED,56))

| ATTRIBUTE | | SOURCE PROGRAM | STATE VECTOR EXECUTION HISTORY | | | | |
|---|---|---|---|---|---|---|---|
| | | | i | $S^i$ | | | |
| NUMBER | NEXT | | | NS | A | B | L |
| | | Program Example2; | | | | | |
| | | . . . | | | | | |
| | | Procedure P(A, B: Integer); | | | | | |
| | | Var L: Integer; | | | | | |
| | | Begin | | | | | |
| 11 | 12 | *P's entry point* | | | | | |
| | | . . . | | | | | |
| | | end; (* Procedure P *) | | | | | |
| | | . . . | | | | | |
| | | Begin | | | | | |
| | . . . | | . . . | | | | |
| | | | 55 | 20,X,55 | newstk | newstk | newstk |
| 20 | 21 | P(30,31) | | | | | |
| | | | | 11,20,56 | | | |
| | | | 56 | 21,20,56 | 30,20,56 | 31,20,56 | I,U,56 |
| | . . . | | . . . | | | | |

- U denotes UNDEFINED.
- X denotes a value that is not relevant to the example.
- Each state vector element is a triple (V, S, N) where
  - V is the content value,
  - S is the number of the statement that assigns value V, and
  - N is the number of the state that follows the value change.
- I is the initial value of the local variable.

**Figure 3. An Example Procedure Call**

where

NS is the index of the element that points to the next statement to execute,

vpindex(S, i) returns the index in state S of value parameter i, for $1 \leq i \leq 2$,

lvindex(S, 1) returns the index in state S of local variable 1, and

ivalue is the initial value of the local variable.

The return from this procedure is modeled by a similar function which POPs the parameters, local variables, and NS. These functions are defined in the next chapter.

## 3.6 A Model for Level 3 Languages

Level 3 languages add pointers and dynamically allocated and deallocated variables to the Level 2 languages. Modeling these features requires two significant changes to the state vector. First, the vector must be able to shrink and grow as variables are allocated and deallocated. This changing size is accomplished by making the state a vector with an infinite number of elements, only a finite number of which are *active*, or currently in use. Introducing pointers into the languages introduces the possibility of wild addressing errors. Modeling these errors requires the second change: that each variable's address in memory be part of the cell associated with that variable. With the addition of an address, each cell in the state vector becomes a 4-tuple. The definitional changes required by these modifications are detailed below.

## 3.6.1 The State Vector

To have addresses as storeable values (for pointers) a set of addresses is defined and integrated into sets analogous to those defined for Level 2 languages. The set of STATES is also modified to reflect its infinite nature. New notation and functions are defined to operate on the newly defined sets.

### *3.6.1.1 ADDRESSES*

Working with pointers requires that the model reflect a computer's ability to access a data value through an address. In a model that does not capture the occurrence and effects of wild address errors, element access can be effected by indices to vector elements, but to model this particular error, actual addresses are required. These addresses are taken from the set

ADDRESSES = $\{ i \mid i \in$ integer, $0 \le i \le$ MAXADDR $\}$

for some maximum address $0 \le$ MAXADDR $\le$ MAXINT,

where MAXINT is the maximum representable integer.

Requiring that addresses be below MAXINT is in keeping with the finite memory assumption and also allows us to model the error of insufficient available memory for dynamic variable allocation.

### *3.6.1.2 VALUES*

The set VALUES has only one addition from Level 1: it now contains the distinguished element NIL, the nil pointer. Because VALUES contains non-negative integers less than or equal to MAXINT, it is a superset of ADDRESSES, and therefore members of ADDRESSES can be assigned to pointers.

### *3.6.1.3 CELLS*

CELLS now contain unique addresses and are 4-tuples defined as

$$\text{CELLS} = \{ (v, i, j, a) \mid v \in \text{VALUES}; i, j \in \text{INTEGER}; a \in \text{ADDRESSES} \}$$

In keeping with the obvious interpretation of these pairs, we call a cell's first member its *contents* and the last member, its *address*. As discussed in section 4.3.2, an unique address is assigned to each cell representing a statically defined variable when the state is initialized and to each cell representing a dynamic variable when that cell is allocated. The function NewCell returns a cell with a newly allocated, unique address and a system assigned initial value. It is defined as

- NewCell: INTEGER $\rightarrow$ CELLS: i $\mapsto$ (ivalue, UNDEFINED, i, addr),
  where i specifies the state at which the cell is allocated, ivalue is the system defined initial value, and addr $\in$ ADDRESSES is the address of a newly allocated cell. Addr must be consistent with the implementation allocation scheme, but we do not attempt to model it further.

### 3.6.1.4 STACKS

This set is the same as defined in Level 2, except that now a stack consists of cells that are 4-tuples and not 3-tuples.

STACKS = { stk | stk is a stack whose members are in CELLS }

The new definition of STATES requires new stack functions. Each function keeps its name and performs a similar operation, only the domains have changed. The same axioms hold and are not repeated here. Assuming

s is an arbitrary member of STACKS and

c is an arbitrary member of CELLS, we define

- newstk: $\rightarrow$ STACKS

  newstk = an empty stack
- addtop: STACKS $\times$ CELLS $\rightarrow$ STACKS

  addtop(s, c) = s with cell c pushed onto it
- deltop: STACKS $\rightarrow$ STACKS

  deltop(s) = s with its top removed
- top: STACKS $\rightarrow$ CELLS

  top(s) = the cell that is on top of s
- isempty: STACKS $\rightarrow$ BOOLEAN

  isempty(s) = (s $\equiv$ newstk)

### 3.6.1.5 *STATES*

At this level, the domain STATES contains infinite rather than finite length vectors of STACKS. This modification is required so that the model can represent the changes that are caused by the dynamic allocation and deallocation of dynamic variables, i.e. by the Pascal statements New and Dispose. At any point in a computation, some state vector elements are associated with program variables and procedure parameters while others are associated with currently allocated dynamic variables. These two groups of elements define the *active* elements. The remaining elements are termed inactive. Thus a call to New (Dispose) is partially modeled by increasing (decreasing) the number of active elements. Formally, we define STATES as

STATES = $\{ S \mid S = <s_1, s_2, \ldots, s_i, \ldots>$, where, for $i \geq 1$, $s_i \in$ STACKS $\}$.

The function

NumActive: STATES $\rightarrow$ INTEGER,

returns the number of active elements in state S. If NumActive(S) = n then elements $s_1, \ldots, s_n$ of S are active and $s_{n+i}$, $i \geq 1$, are not. For convenience we denote S by

$<s_1, \ldots, s_n>$

and ignore its infinite number of inactive elements.

The following polymorphic functions provide access to the components of element number e of state vector S.

- Contents(S, e) returns $S_e|_1$ for Level 1 languages and top$(S_e)|_1$ for Level 2 and Level 3.
- Stmt(S, e), returns $S_e|_2$ for Level 1 languages and top$(S_e)|_2$ for Levels 2 and 3.

- State(S, e), returns $S_e|_3$ for Level 1 languages and $top(S_e)|_3$ for Levels 2 and 3.
- Address(S, e) returns $top(S_e)|_4$ Level 3 languages and is undefined for Level 1 and Level 2.

When called with an arbitrary state vector and any element except NS (the Next Statement pointer) as arguments, these functions return the following information.

- Contents(S, e) returns the content value of the element, i.e. the value of the variable associated with element e,
- Stmt(S, e) returns the sequence number of the statement that assigned the element its value,
- State(S, e) returns the sequence number of the state at which the value was assigned, and
- Address(S, e) returns the address allocated to the variable associated with element number e.

In contrast, when accessing element NS (the Next Statement pointer) these functions provide the following information.

- the sequence number of the next statement to be executed, using Contents(S, NS),
- the sequence number of the statement that has just been executed, using Stmt(S, NS), and
- the sequence number of the current state, using State(S, NS).

The inclusion of addresses in cells makes it possible for a wild address operation to access any cell (except NS) in the state vector, given the cell's address. Therefore we define the function FindAddr that locates the cell with a given address. Since the located cell may not be on the top of its stack, FindAddr returns the number of the element (stack) containing the cell and the position of the cell in that element, where the top cell of a stack is at position one. Thus we define

- FindAddr: STATES × ADDRESSES → INTEGER × INTEGER

  FindAddr(S, addr) = (e, p), where addr = $c|_4$, for cell c in position p of element e of state vector S. FindAddr returns (UNDEFINED, UNDEFINED) if there is no such cell.

### 3.6.2 Action Functions

Level 3 necessitates new action functions that cause the state vector to grow and shrink as dynamic variables are allocated and deallocated. Its stack functions are analogous to those defined at Level 2, but the REPLACE function must be redefined to accommodate the modeling of wild addressing. In the definitions below,

S $\in$ STATES is the infinite vector $<s_1, \ldots, s_n>$,

(we omit the non-active elements) where $s_i \in$ STACKS, for $1 \leq i$, and NumActive(S) = n. The two new functions to be defined are ADD and DEL.

- ADD(S, m) returns a state vector T that is a copy of S up to the number of active elements of S, and that has m additional active elements. Therefore,

  NumActive(T) = NumActive(S) + m = n + m.

  Each new cell is allocated through

  NewCell(State(S, NS) + 1)

  providing an initial cell configuration of (ivalue, UNDEFINED, State(S, NS) + 1, newaddress).

In each such cell, State(S, NS) is the number of the state preceding the allocation statement and thus State(S, NS) + 1 is the number of the first state that contains the newly allocated cells. The definition of ADD, then, is

ADD: STATES × INTEGER → STATES: $(S, m) \mapsto < t_1, ..., t_{n+m} >$

where $t_i = s_i$, for $1 \leq i \leq n$

$t_j$ = addtop(newstk, NewCell( State(S, NS) + 1 )), for $n+1 \leq j \leq n+m$

- DEL(S, i, m) returns a state vector T that is a copy of S except that the m elements (i+0) to (i+m-1) of S have been made inactive and elements (i+m) to n are shifted "down" and remain active. Therefore, NumActive(T) = NumActive(S) - m = n - m. Formally,

DEL: STATES × INTEGER × INTEGER → STATES: $(S, i, m) \mapsto < t_1, ..., t_{n-m} >$,

where $t_j = s_j$, for $1 \leq j \leq i\text{-}1$

$t_{k-m} = s_j$, for $i+m \leq k \leq n$

Remember that $< t_1, ..., t_{n-m} >$ denotes the active elements in the infinite state vector $< t_1, ..., >$. The following functions are analogous to those defined at Level 2.

- PUSH'(S, i, c) returns a state vector that is a copy of S except that c ∈ CELLS is pushed onto element $S_i$ of S. This function is analogous to PUSH defined at Level 2 except that it operates on stacks of 4-tuples and that the STATES set now contains infinite state vectors.

PUSH′: STATES × INTEGER × CELLS → STATES: $(S, i, p) \mapsto <t_1, ..., t_n>$,

where $t_j = s_j$ for $1 \leq j \leq n, j \neq i$,

$t_i = \text{addtop}(s_i, p)$

- POP′ is analogous to the POP defined at Level 2 except that it operates on stacks of CELLS and not VALUES, and that the STATES domain now contains infinite state vectors.

POP′: STATES × INTEGER → STATES: $(S, i) \mapsto <t_1, ..., t_n>$,

where $t_j = s_j$ for $1 \leq j \leq n, j \neq i$

$t_i = \text{deltop}(s_i)$.

- Assignments in Level 2 languages replace only the top cell of a stack, but at Level 3 an assignment that causes a wild address error can necessitate a modification to any cell of a stack. REPLACE″, the replace function for Level 3, can therefore modify a cell at any position in the designated element. That is, REPLACE″ (S, e, p, c) puts the cell c in the place of the cell at position p (where position 1 is the top) of element e of state vector S. REPLACE″ then is defined as

REPLACE'': STATES × INTEGER × INTEGER × VALUES → STATES:

$(S, e, p, c) \mapsto \langle t_1, \ldots, t_n \rangle$,

where $t_j = s_j$ for $1 \le j \le n, j \ne e$

$t_e$ = addtop( addtop( ... (addtop(addtop(deltop$^p$($s_e$), c),

top(deltop$^{p-2}$($s_e$))), ... top(deltop($s_e$))), top($s_e$) ).

To make this complex function manageable we use the notation

deltop$^n$(s) = deltop(deltop$^{n-1}$(s)) and deltop$^1$(s) = deltop(s).

To replace the cell at position p, this function simply pops the top p cells, pushes on the new cell, and then reconstructs that part of the stack that was popped off.

## 3.6.3 An Example State Transition Function

To close this chapter, this subsection illustrates the use of Level 3 action functions in the definition of a Level 3 state transition function that models the allocation of a two component dynamic variable. Figure 4 on page 47 shows the transition caused by the execution of this statement. This transition is modeled by the function

$\lambda$S.REPLACE''(S, NS, (21, 20, 56, UNDEFINED) )

o $\lambda$S.REPLACE''(S, [P], 1, (96, 20, 56, 91) )

o $\lambda$S.ADD(S, 2)

In this example, cells contain addresses, and the number of active elements in the state varies. Because NS does not correspond to any memory location, its address has the value UNDEFINED. Execution of the procedure new allocates two compo-

| ATTRIBUTES | | SOURCE PROGRAM | STATE VECTOR EXECUTION HISTORY | | | | |
|---|---|---|---|---|---|---|---|
| | | | i | $S^i$ | | | |
| NUMBER | NEXT | | | NS | P | A | B |
| | | Program Example3; | | | | | |
| | | Type List=↑Node | | | | | |
| | | Node=Record | | | | | |
| | | val:Integer; | | | | | |
| | | next:List; end; | | | | | |
| | | Var P: List; | | | | | |
| | | A, B: Integer; | | | | | |
| | | . . . | . . . | | | | |
| | | | 55 | 20,X,55,U | $I_1$,U,0,91 | | |
| 20 | X | new(P); | | | | | |
| | | | 56 | X,20,56,U | 96,20,56,91 | $I_2$,U,56,96 | $I_3$U,56,97 |
| | | . . . | . . . | | | | |
| | | End. | | | | | |

- U denotes UNDEFINED.
- X denotes a value that is not relevant to the example.
- Each state vector element is a 4-tuple (V, S, N, A) where
  - V is the content value,
  - S is the number of the statement that assigns V,
  - N is the number of the state that follows the assignment, and
  - A is the address of the variable associated with the 4-tuple.
- $I_1$, $I_2$, $I_3$ are the initial values of the pointer and the newly allocated cells.

**Figure 4. An Example Dynamic Variable Allocation**

nents with addresses 96 and 97, causing P to be assigned the value 96 and the number of active elements of the state to grow by 2, i.e.

i.e. NumActive($S^{56}$) = NumActive($S^{55}$) + 2.

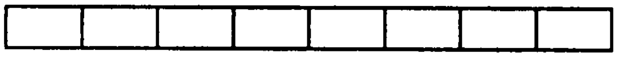
This example concludes the description of the state vectors and action functions used to model Level 1, Level 2, and Level 3 languages. The following section summarizes characteristics of the model.

## 3.7 Summary

In summary, Figure 5 presents an overview of the state vector structure and the action functions for each level of the language hierarchy. At Level 1, the fixed length state vector has 3-tuples as elements, and the action function REPLACE modifies these elements. At Level 2 the vector length remains fixed, but the elements are now stacks of 3-tuple cells. The action functions PUSH and POP add and remove cells from these stack elements, and REPLACE' modifies the 3-tuple that is on top of a specified stack. At the most complex level, 3, a state is of infinite length and has stacks of 4-tuples as elements. The functions ADD and DEL cause elements to become active and inactive, and PUSH' and POP' modify the stack elements as they do at Level 2. REPLACE'' is redefined so that it can access any cell of a stack, not just the top.

This completes the definition of the state vectors and the action functions that operate on them. The following chapter describes how Level 1, 2 and 3 statements and programs are modeled using state vector execution histories.

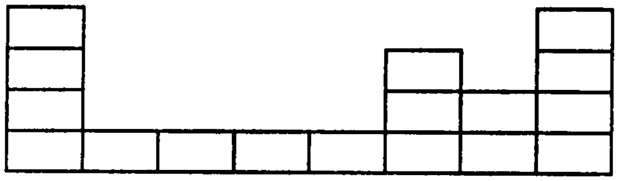
***LEVEL 1***

State Vector: 

Elements are triples: (Value, Statement, State)

Action Function: REPLACE(S, e, c) replaces in element $S_e$ with triple c.

***LEVEL 2***

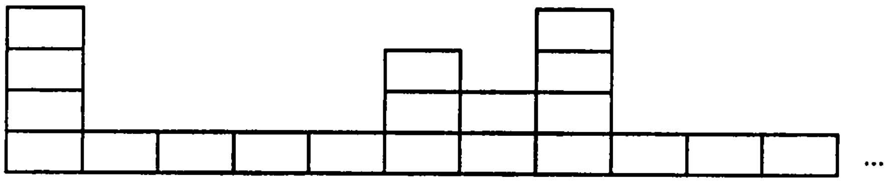
State Vector: 

Elements are stacks of the triples defined above

Action Functions: PUSH(S, e, c) pushes cell c on element $S_e$.
POP(S, e) pops the top value on element $S_e$.

REPLACE'(S, e, c) replaces the top cell of element $S_e$ with c.

***LEVEL 3***

State Vector: 

Elements are stacks of 4-tuples: (Value, Statement, State, Address)

Action Functions: ADD(S, m) makes m additional elements of S active
DEL(S, i, m) makes elements i to (i+m-1) of S inactive and shifts down any higher active elements.

PUSH'(S, e, c) pushes cell c on element $S_e$.
POP'(S, e) pops the top element from stack element $S_e$.

REPLACE''(S, e, p, c) replaces the cell at position p of element $S_e$ with cell c. Position 1 is the top of the stack.

**Figure 5. Model Summary**

# 4.0 Execution History Modeling of Statement and Program Execution

A state vector represents a snapshot of a program's condition at a particular point in a computation. The transition from a single state vector to its successor represents the execution of a single program statement, and thus a sequence of vectors represents the execution of several statements. This chapter defines the state vector to state vector transitions that represent execution of statements from each of the language levels and that ultimately define the state vector *sequence* that represents execution of an entire program. Section 4.1 describes the modeling of Level 1 languages and includes an informal description of the statements and data types of these languages and a definition of the state vector to state vector transition functions that model them. The following two sections similarly define Level 2 and Level 3 languages and the transition functions that model them. The final section of this chapter describes the execution histories that model program execution and describes a scheme for classifying these execution histories.

## 4.1 Modeling Level 1 Languages

This section defines the state transition functions that model Level 1 language statements. Preceding this definition are subsections that describe Level 1 languages and techniques for modeling control flow.

### 4.1.1 Language Description

The focus of this research is on developing debugging techniques for programs that are written in imperative, strongly typed, procedure oriented languages. Therefore, in defining the language classes to consider, the programming language Pascal [WIRN71, IEEE83a] serves as a model because it is well known and because it has features that are common to modern languages but are not trivial. To facilitate communication, we use the syntax of Pascal throughout. In this section we describe a typical Level 1 language. Except where noted, this description also applies to Level 2 and Level 3 languages since these classes contain the Level 1 languages.

#### *4.1.1.1 Types*

A program written in this typical language consists of a sequence of declarations followed by a sequence of statements. Each variable used in the statement sequence must appear exactly once in a type declaration where it is declared to be of simple or composite type. The simple types are:

- Integers in the range [MININT..MAXINT],
- Subranges of the integers,
- Boolean values,
- Floating point numbers and
- Characters.

The composite types are

- Records whose components are of simple type and
- Single dimension arrays whose elements are of simple type and and whose selectors are of integer subrange type.

We assume the language is strongly typed. The resolution of the ambiguities of strong typing [WELJ77], such as the compatiblity of integer subranges, follows the IEEE Pascal Standard [IEEE83a]. Each variable, as well as each local variable and each dynamically allocated variable, takes some system defined initial value.

#### *4.1.1.2 Statements*

The second portion of a program consists of a sequence of simple and/or compound statements. The only simple statement in Level 1 languages is the assignment statement. Other simple statements are needed for the procedures and dynamic variables of Level 2 and Level 3 languages. The compound statements are

- the while statement and
- the if statement.

Each compound statement consists of a logical expression and one or two embedded statement sequences. The logical expression is called the statement's *test*. We use the term *operation* to refer to either a simple statement or the test part of a compound statement. Each operation is modeled by a single state to state transition.

An assignment statement modifies the value of a single state vector element by replacing that value with another. The while statement executes the statement(s) in its embedded sequence as long as the test is true, and the if statement executes the statements embedded in one of its two branches, based on the value of the test. Statement sequences are compositions of one or more simple or structured statements and are executed in textual order.

To avoid unnecessary detail, we do not define the exact set of operators that can be used to construct expressions. We assume that the set contains the usual arithmetic and boolean operators. In addition, some set of system defined functions is available. Each of these functions operates on one or more data values to calculate another value. As discussed in subsection 4.2.8, user defined functions are excluded from the model, but because the system functions are assumed to operate in a single, indivisible step, with no visible change in the flow of control, they are included.

One of the goals of the model is that a single transition from one state to its successor should represent the execution of a single simple program statement. For compound statements, the transition should reflect only the transfer of control that results from evaluating the statement's boolean condition, and not the execution of the embedded statements. The functions that model compound statements therefore model only the test of the statement, and not its embedded statements. This creates a medium grain EH rather than one of course grain in which a transition reflects the action of an entire compound statement, or one of fine grain one in which a transition reflects each action involved in evaluating expressions within a statement.

Since we are interested in a medium grain EH we do not model the mechanism that evaluates expressions within statements; each state transition function uses selector functions that return the value(s) of its constituent expression(s) as needed. We do not define how these selector functions determine the expression values, but simply assume that they compute the same values as the processing environment. In statements that abnormally terminate it is necessary to determine the exact component of the statement that causes the error. Therefore Chapter 5 defines an error substantiation function that evaluates an erroneous statement and finds the expression causing the error.

The three language classes use a set of statements which is sufficiently large that the techniques and understanding developed are general enough to have real application, but it is also sufficiently small that programs can easily be analysed and that the focus of the research can remain on understanding EH based debugging rather than on defining many highly specific techniques. The remainder of this section discusses some language features that are not included in the model.

The most noticable absence is the lack of input and output statements. Including these statements would make the model more complex, thereby making EH analysis and the presentation of results more difficult. It would also add nondeterminism to a program's execution which would make certain techniques inapplicable. The model modifications required by I/O and the additional techniques it would entail are the subject of future work. User defined functions are also excluded from the model because we do not want to consider transfer of control within statements. This is discussed in more detail in section 4.2.8. Another statement not included in the language class is the go to. This statement can be represented within the current model, but it is excluded because programs written without the go to are usually considered to be easier to debug. Other language features are excluded to keep this study focused on developing an understanding of execution history analysis and debugging, rather than on developing a variety of specialized techniques. These features include

- Alternate control structures (for and repeat),
- Enumerated types,
- Sets,
- Variant records,
- Arrays of records, and
- Records of arrays.

This discussion of excluded language features concludes the overview of Level 1 languages. The definition of the state transition functions that model execution of Level 1 statements is preceded by a discussion of modeling control flow.

### 4.1.2 Modeling Control Flow

The overview in section 3.3 provides the necessary background for modeling control flow by using element NS of the state vector and the operation attributes NUMBER, NEXT, TD and FD. An operation, remember, is either a simple statement or the test of a compound statement. In review, the attribute NUMBER gives a unique sequence number to each operation, NEXT specifies the number of the operation to execute after a simple statement, and TD (True Destination) and FD (False Destination) provide the numbers of the operations to execute if a test evaluates to true or false, respectively. A transition that models an operation replaces state vector element NS with the value of the appropriate attribute so that $S_{NS}$ indicates the next operation to execute. This causes the state vector sequence to be a record of the statements executed in a program.

In the example shown in Figure 2 of Chapter 3 statements 1 and 2 have a NEXT attribute of 2 and 3 respectively, and thus, as shown in element NS of states 1 and 2, control flows from statement 1 to statement 2 and then to statement 3. Statement 3's TD and FD attributes indicate that control flows to statement 4 when the IF statement is true and statement 5 when false. In state 4, element NS's value shows the choice of the false destination and in state 5 the value UNDEFINED indicates that statement 6 is the program's last statement and that the computation should halt.

In a program with structured statements, the hierarchy of statements expresses the underlying structure of the algorithm. The preceding method of modeling control flow effectively flattens out this hierarchy and distills the algorithm's structure into the attributes of the program's operations. Therefore, a crucial part of modeling control flow is assigning the value of the attributes so that they reflect the structure implied by the compound statements. For a specific language, an attribute grammer [KNUD68, WATD79] effectively calculates the values for these attributes. The (simplified) attribute grammar shown in Appendix A demonstrates this calculation for a hypothetical Level 1 language.

To model the transition caused by an operation, a state transition function must access these attributes. Access is provided by the following functions which extract the values of these attributes from the operation whose sequence number, i.e. NUMBER attribute, matches the value of state vector element NS, i.e. they extract the attribute values from the operation to which element NS points.

**next(S)** Returns the attribute designating the next statement to be executed relative to the operation whose sequence number matches element NS of state S.

**td(S)** Returns the true destination attribute of the operation whose sequence number matches element NS of state S.

**fd(S)** Returns the false destination attribute of the operation whose sequence number matches element NS of state S.

The state transition functions that model the Level 1 statements (operations) are defined in the next sections.

## 4.1.3 Modeling Assignment

The assignment statement

*identifier* := *expression*

assigns the value of *expression* to the variable specified by *identifier*. If the identifier is an array reference then the statement assigns the value to the specified array member. The action of this statement is expressed by the function

**ASSIGN**: STATES $\rightarrow$ STATES

**ASSIGN** $\equiv \lambda S.\mathrm{REPLACE}(S, NS, (\mathrm{next}(S), S_{NS}|_1, S_{NS}|_3 + 1))$

$\quad\quad\quad\quad\quad \circ\ \lambda S.\mathrm{REPLACE}(S, \mathrm{iden}(S), (\mathrm{expr}(S), S_{NS}|_1, S_{NS}|_3 + 1))$

where

iden(S) returns the index of the element of S that corresponds to the identifier and expr(S) returns the value of the expression evaluated under state S.

The first action of ASSIGN is to replace the value of the element of S that corresponds to identifier with the value of the expression, both as evaluated under state vector S. The effect of the assignment statement is completed by replacing the element NS with the value of the assignment's successor attribute, causing control to flow from the assignment to the statement that is to be executed after the assignment's execution is complete. In both replacements, the second member of the new triple is the number of the statement that is currently being executed, i.e. $S_{NS}|_1$, and the third member is the number of following state, $S_{NS}|_3 + 1$. Element NS is replaced in every transition and thus $S_{NS}|_1$, and $S_{NS}|_3$ always indicate the current statement and state respectively until element NS is modified.

## 4.1.4 Modeling While Statements

The while loop serves as an iteration statement for Level 1 languages. This statement has two parts: a loop test and a loop body. The body is a sequence of statements that is executed repeatedly as long as the loop test is true. If the test is initially false, then the body is never executed. Modeling the execution of a while statement requires modeling the conditional transfer at the loop test and also the unconditional transfer from the last statement of the loop body back to the test. Therefore the test's true destination attribute is the sequence number of the first statement in the loop body, while the false destination attribute is the sequence number of the statement that is to be executed after the loop's execution is complete, as determined by the structure in which the loop occurs. The transfer of control flow from the last statement of the loop back to the loop test is modeled by making the successor of this last statement the while statement itself, i.e the loop test. As the loop body completes execution, element NS is replaced by the last embedded statement's successor number (i.e. by the sequence number of the loop's condition) and control flows back to the test. The test of the while statement is modeled by the function

---

**WHILE**: STATES $\rightarrow$ STATES

**WHILE** $\equiv \lambda$S.REPLACE(S, NS, (cond( bvalue(S), td(S), fd(S) ) , $S_{NS}|_1$, $S_{NS}|_3 + 1$ ) )

where bvalue(S) returns the boolean value of the test of the while statement, and cond(a, b, c) returns b or c if a is true or false, respectively.

---

The values $S_{NS}|_1$ and $S_{NS}|_3$ are used as in the function ASSIGN.

## 4.1.5 Modeling If Statements

The if statement has three parts: a test, a then branch and an else branch. When the test is true, the then branch is executed; when false, the else. In either case, after the branch, flow of control joins at the statement that follows the if statement. Modeling the execution of an if statement requires modeling both the conditional transfer at the test and also the unconditional transfer from the last statement embedded in the true branch to the statement following the if. Therefore the test's true destination attribute is the sequence number of the first statement in the then branch, while the false destination attribute is the sequence number of the first statement in the else branch.

The transfer of control from the then or else branch to the statement after the if relies on the successor number of the last statement of each branch. In each case, this attribute points to the statement that is to be executed after the if statement is completed. As the appropriate last statement completes its execution, element NS is replaced by the value of this attribute and control flows to the next statement. Thus the successor numbers calculated using attribute grammars in a static analysis of the program provide the information necessary to model the dynamics of flow of control. The test of the if statement is modeled by the function

---

**IF**: STATES $\rightarrow$ STATES

**IF** $\equiv \lambda$S.REPLACE(S, NS, (cond( bvalue(S), td(S), fd(S) ) , $S_{NS}|_1$, $S_{NS}|_3 + 1$ ) )

where bvalue(S) returns the boolean value of the test of the if statement, and

cond(a, b, c) returns b or c if a is true or false, respectively.

---

The values $S_{NS}|_1$ and $S_{NS}|_3$ are used as in the function ASSIGN.

### 4.1.6 Modeling Other Control Structures

Level 1 languages have the three control structures sequence, selection and iteration. The first of these is reflected implicitly in statement sequences, and the second and third are the if and while statements. While we have restricted our modeling to these three structures, the work of Boehm and Jacopini [BOEC66] shows the sufficiency of these three structures for modeling programs built from arbitrary control structures. The preceding three sections have shown the representation of these structures in our model, and hence, although the languages considered are restricted, the model is sufficiently powerful to represent arbitrary structures.

## *4.2 Modeling Level 2 Languages*

Level 2 languages, provide all of the data types and statements found in Level 1 languages plus procedure definition and execution. Modeling these languages requires the state vector characteristics defined in the previous chapter and two additional state transition functions: CALL and RETURN. Before describing these additional functions, the following subsection describes the semantics of procedures and discusses how they are modeled.

### 4.2.1 Informal Semantics

We assume that procedures in Level 2 languages are similar to procedures in the programming language Pascal. Specifically, parameters are passed either by value, with the actual value being passed, or by reference, with the address of the parameter being passed. Parameters must follow the rules of assignment compatibility as defined in the IEEE Pascal standard. Programs are block structured and static scope rules apply; variables exist only during the lifetime of the procedure that declares them. Local variables take some system defined initial value on procedure entry. Side effects are manifested through reference parameters and non-local variables. Unlike Pascal, procedures may not be parameters of other procedures and user defined functions may not be used for the reasons discussed in section 4.2.8. However, mutually recursive procedures are allowed. Each procedure is assumed to have a single, well defined entry point. Invocation of the procedure causes control to transfer to this entry point and a return from the procedure causes control to transfer to the statement that is the logical successor of the calling statement (i.e. to the statement indicated by the call's NEXT attribute). The return may be implicit in the procedure rather than directly specified with a return statement. Based on this informal semantics, the following subsections discuss the modeling of Level 2 languages.

### 4.2.2 The State Vector

As discussed in Chapter 2, modeling Level 2 languages requires a state vector whose elements are stacks. This modification allows the state vector model to reflect the stacking and popping of local variables, value parameters and procedure entry points and return addresses. Without mutually recursive procedures, the maximum depth of each stack could be calculated and fixed length stacks could be used. However, since we allow mutually recursive procedures, stacks of arbitrary depth are required.

As in the Level 1 languages, each global variable in a Level 2 program is allocated one element (or more for records) in the state vector. In addition, state vector elements are allocated for each local variable and each formal value parameter declared in a Level 2 program. Elements that are associated with local variables that are declared in procedures which have no current activation exist as empty stacks. Invoking a procedure causes initial values to be placed on these stacks. The stack characteristics of element NS allow it to be used to model the flow of control of the procedure call and return.

### 4.2.3 Flow of Control

As in Level 1 languages, flow of control is modeled using state vector element NS in conjunction with attributes of simple statements and compound statement tests. Simple statements now include procedure call and return. As in Level 1, attributes include a unique sequence number for each simple statement and test, a unique successor for each simple statement, and true and false destinations for each test operation. Each statement in a procedure is given a unique sequence number. If the procedure's return is not explicitly coded, then the implicit return is also given a number. To model the transfer to and from a procedure, execution of a procedure invocation first causes the return address, that is, the number of the calling statement's logical successor, to be stacked on element NS, and then causes the number of the procedure's entry point to be stacked. The transfer back to the return address is modeled by popping the top element from element NS, leaving the return address on top of the element NS stack.

### 4.2.4 Reference Parameters

No additional action is needed to model parameters that are passed by reference. These reference parameters can be ignored because they are actually veiled references to some local or global variable. Since we assume the semantics of reference parameters to be like that of Pascal, where each change of a reference parameter is directly and immediately reflected, any change that affects the parameter is reflected in the variable to which it refers. This approach shifts the responsibility of interpreting reference parameters to the error analysis routines. If a routine needs to know a procedure's actual parameters at a particular point in a computation, then it examines the code and the EH to determine the elements that correspond to the actual parameters. This process is discussed in more detail in sections 5.3.3.1. The following subsections describe the state transition functions that model a procedure call, and a procedure return.

### 4.2.5 Modeling Procedure Call

We assume that the procedure whose invocation and return are being modeled has

- M elements allocated to value parameters and
- N elements allocated to local variables.

A procedure invocation causes the following modifications to the current state:

- A triple representing some system defined initial value is pushed onto each of the N elements that correspond to local variables.
- Triples representing the M values of the value parameters are pushed onto their respective elements,
- The return address triple is stacked on element NS, and
- The triple representing the entry point of the called procedure is stacked on element NS.

These actions are modeled by the following function:

---

CALL: STATES $\rightarrow$ STATES

CALL $\equiv$ $\lambda$S.PUSH(S, NS, (ep(S), top(deltop($S_{NS}$))$|_1$, top($S_{NS}$)$|_3$ ) )

o $\lambda$S.PUSH(S, NS, (ra(S), top($S_{NS}$)$|_1$, top($S_{NS}$)$|_3 + 1$ ) )

o $\lambda$S.PUSH(S, vpindex(S, M), (vp(S, M), top($S_{NS}$)$|_1$, top($S_{NS}$)$|_3 + 1$ ) ) o ...

o $\lambda$S.PUSH(S, vpindex(S, 1), (vp(S, 1), top($S_{NS}$)$|_1$, top($S_{NS}$)$|_3 + 1$ ) )

o $\lambda$S.PUSH(S, lvindex(S, N), ($ivalue_N$, UNDEFINED, top($S_{NS}$)$|_3 + 1$ ) ) o ...

o $\lambda$S.PUSH(S, lvindex(S, 1), ($ivalue_1$, UNDEFINED, top($S_{NS}$)$|_3 + 1$ ) )

where

ep(S) returns the entry point of the procedure being called,

ra(S) returns the return address,

vpindex(S, i) returns the index in state S of value parameter i, for $1 \leq i \leq M$,

vp(S, i) returns the value in state S of value parameter i, for $1 \leq i \leq M$,

lvindex(S, i) returns the index in state S of local variable i, for $1 \leq i \leq N$.

$ivalue_i$ is the initial value of local variable i, for $1 \leq i \leq N$.

---

Each value that is placed on a stack is a triple whose second element either points to the call statement or has the value UNDEFINED, and whose third component is the number of the following state. Each local variable's second element, UNDEFINED, indicates that no assignment has yet been made to that local variable. When the procedure's entry point is pushed, element NS has already been modified so that the number of the call statement is directly below the top and the number of the next state is in the top triple. Therefore the second and third elements of that 3-tuple are $top(deltop(S_{NS}))|_1$ and $top(S_{NS})|_3$.

### 4.2.6 Modeling Procedure Return

We assume that a return statement exists in each routine. In a language that does not contain an explicit return the model can simulate one by causing the successor of the last statement of each routine to be an implicit return. A procedure return causes the following modifications to the state vector:

- Each of the M value parameter elements is popped,
- Each of the N local parameter elements is popped and
- Element NS is popped, leaving the return address on top of element NS.

These actions are modeled by the following function:

---

**RETURN**: STATES $\rightarrow$ STATES

**RETURN** $\equiv \lambda$S.POP(S, NS)

o $\lambda$S.POP(S, vpindex(S, M)) o ... o $\lambda$S.POP(S, vpindex(S, 1))

o $\lambda$S.POP(S, lvindex(S, N)) o ... o $\lambda$S.POP(S, lvindex(S, 1)) where

vpindex(S, i) returns the index in state S of value parameter i, for $1 \leq i \leq M$.

lvindex(S, i) returns the index in state S of local variable i, for $1 \leq i \leq N$.

---

Popping the local variables and value parameters cause the procedure invocation to be transparent except for the side effect modifications of the non-local variables and reference parameters.

### 4.2.7 Modeling the Level 1 Statements

For completeness the state transition functions that model the Level 1 statements assignment, while, and if are redefined using Level 2 states and REPLACE′, the Level 2 replacement function. The redefined functions are given below, without additional comment.

**ASSIGN**: STATES $\rightarrow$ STATES

**ASSIGN** $\equiv \lambda$S.REPLACE′(S, NS, (next(S), top($S_{NS}$)$|_1$, top($S_{NS}$)$|_3 + 1$ ) )

o $\lambda$S.REPLACE′(S, iden(S), (expr(S), top($S_{NS}$)$|_1$, top($S_{NS}$)$|_3 + 1$ ) )

where

iden(S) returns the index of the element of S that corresponds the identifier and expr(S) returns the value of the expression evaluated under state S.

**WHILE**: STATES → STATES

**WHILE** ≡ $\lambda$S.REPLACE'(S, NS, (cond(bvalue(S),td(S),fd(S)), top($S_{NS}$)$|_1$, top($S_{NS}$)$|_3$ + 1))

where bvalue(S) returns the boolean value of the test of the while statement, and cond(a, b, c) returns b or c if a is true or false, respectively.

**IF**: STATES → STATES

**IF** ≡ $\lambda$S.REPLACE'(S, NS, ( cond(bvalue(S), td(S), fd(S) ), top($S_{NS}$)$|_1$, top($S_{NS}$)$|_3$ + 1 ) )

where bvalue(S) returns the boolean value of the test of the if statement, and cond(a, b, c) returns b or c if a is true or false, respectively.

### 4.2.8 Functions

User defined functions are not included in our model because modeling the execution of statements that contain function calls is impossible within the constraint of modeling execution of a single statement as a single state transition. A function containing statement can not be modeled as a single transition because invocation of the function requires a transfer of control which is itself modeled by a transition. Although we do not include the modeling of user defined functions in our current research effort, one possible way to incorporate them is illustrated in the example below. To model

```
A := F(X) + 5 * F(Y)                                    (* 1 *)
```

where F is defined as

```
function F(P:integer): integer;
  ...
  F := expression
```

is to introduce three new integer variables, F1, F2, and TEMP and modify the function as follows

```
procedure F(P:integer; RV:integer);
  ...
  RV := expression.
```

With these modifications, the following program fragment represents the operation sequence required by the parsing framework to model statement 1.

```
F(Y, F2);
TEMP := 5 * F2
F(X, F1);
A := F1 + TEMP
```

We could introduce this transformation into the model by introducing a new state vector element for each function invocation and transforming function invoking expressions into a series of calls, however this method requires modification of the source program which we want to avoid. Therefore we exclude user defined functions from the model.

## *4.3 Modeling Level 3 Languages*

Level 3 languages extend Level 2 languages with dynamically allocated and deallocated variables and variables of pointer type. As discussed in the previous chapter, modeling these features requires the state vector to have an infinite number of elements, each of which is a stack of cells that contain addresses. In addition it requires the modeling of two new statements, new and dispose, that perform dynamic variable allocation and deallocation. An informal discussion of the semantics of these features and a definition of the state transition functions that model the new statements are the topics of this section.

## 4.3.1 Informal Semantics

We closely follow the language Pascal in defining the semantics of pointers and dynamically allocated variables. First we introduce a new simple type *pointer to structure*. Variables of this type, called *pointers*, have as values addresses of dynamically allocated variables. A pointer is said to *point* to a structure whose address the pointer holds. If a pointer has the value nil, it points to no structure at all. A pointer can legally point only to variables of the type specified in its declaration. Accessing the variable to which a pointer points is called *dereferencing* the pointer. A dynamically allocated variable is created by the system procedure **new**(P) which has the following actions: (1) it allocates a structure of the type to which P can point, (2) it assigns a value to P so that it points to the newly allocated structure, and (3) it leaves the structure itself with a system defined initial value. The structure created by the procedure **new**(P) is called an *identified variable* and P is said to *identify* such a variable. The system procedure **dispose**(P) deallocates the structure pointed to by P, effectively destroying the structure so that it can no longer by accessed. **Dispose**(P) has no effect on P. Any attempt, using P or any other pointer, to access a structure after it has been deallocated is assumed to cause an abnormal termination.

Structures created within a procedure are not affected by a return from that procedure, but any pointer that is local to the procedure disappears when that procedure is exited. Thus, access to structures may be lost on procedure return, but if a non-local variable points to such a structure, then the structure can still be accessed. If an identified structure can not be accessed then it is called *garbage*. Garbage can be created by a procedure return, or more simply, by changing the only pointer that points to an object. The following code illustrates the latter method.

```
new(P);
P := nil
```

The presence of garbage does not create an abnormal termination, but it is a detectable anomalous condition.

Another pointer error, the *dangling reference*, occurs when a pointer's value is the address of no cell in the program's address space. This condition can be caused by a code sequence like

```
new(p);
q := p;
dispose(q)
```

which leaves p as a dangling reference. Actually, **dispose** always causes a dangling reference that exists until the disposed pointer is made to point to another structure or is set to nil. A dangling reference does not cause an abnormal termination, but dereferencing a dangling reference does.

Although a pointer can legally point only to a variable of its declared type, a pointer can illegally point to a identified variable of the wrong type if that identified variable has been allocated at the address at which a dangling reference points. A variable that has a value that is not of the correct type is said to be *incompatible*; thus a pointer is incompatible if it points to an an element of the wrong type or, by default, if it is a dangling reference. The dereferencing or disposing of an incompatible pointer is known as a *wild address* operation; an incompatible pointer is also known as a *wild pointer*. A wild address operation causes an abnormal termination only if it uses an address that is outside the program's address space, as defined by the processing environment. If it uses an address that is in the program's address space then it causes an error that may or may not affect termination. A variable whose value is assigned by a wild address write operation is said to be *corrupt*, and a wild address write operation that corrupts a pointer may also make that pointer incompatible. The following program fragment illustrates a dangling reference that is made incompatible and then is used in a wild address operation.

```
TYPE   prec1 = ↑rec1;
       prec2 = ↑rec2;
       rec1 = record a: integer; b: prec1   end;
       rec2 = record a: prec2;   b: integer end;
VAR    P: prec1;
       Q: prec2;
  . . .
  new(P);
  dispose(P);              { P is now a dangling reference }
  new(Q);                  { Assume this allocation sets Q equal P }
                           { P is now incompatible}
  Q↑.b := 99;              { This operation is legal }
  P↑.b↑.a := 0             { This is a wild address operation }
```

Assuming that pointers and integers have the same word length, the final assignment uses 99 as an address and is a wild address operation. If address 99 can not be accessed, then this operation causes an abnormal termination. If 99 can be accessed, then the variable at address 99 becomes corrupt, but this error is not detected by the processing environment.

### 4.3.2 The State Vector

As described in Chapter 2, modeling dynamic allocation and deallocation requires an infinite state vector in which the number of currently active elements can change. Modeling wild address errors also requires that actual addresses be used in the cells. Using addresses ties the model to the processing environment that is used to run the program which adds needed realism, but also introduces additional complexity to our model.

State vectors, then, have as elements stacks of 4-tuple cells in which the fourth member of the tuple is the address allocated to the variable represented by the cell. As shown in Figure 6, the state vector elements can be divided into four groups, based on when addresses are allocated to cells of each group.

1. Group 1 contains element NS. Its cells do not have addresses, and their fourth element always has the value UNDEFINED.
2. Group 2 contains the elements that are allocated to global variables. Cells in these elements are all allocated addresses when the computation begins.

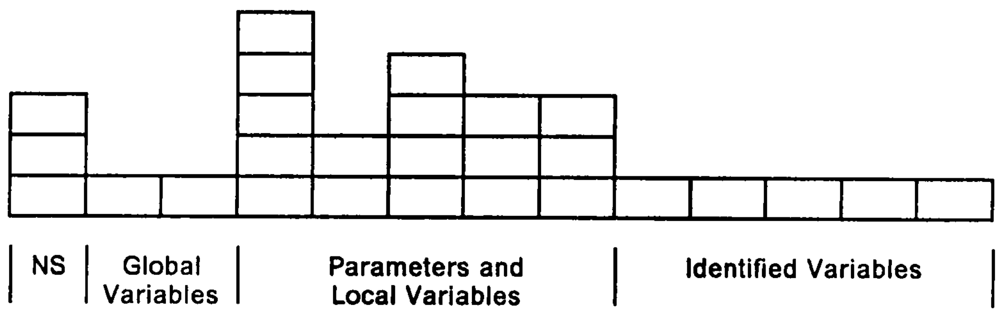


**Figure 6. State Vector Element Groups**

3. Group 3 contains the elements that are allocated to the value parameters and local variables found in procedures. Cells in these elements are allocated addresses when the appropriate procedure is entered.
4. Group 4 contains elements associated with identified variables. Cells in these elements are allocated addresses when the element is allocated and consequently becomes active.

Another characteristic of these groups is that only elements in groups 1 and 3 can contain more than one cell. Elements of groups of 2 and 4 are stacks of one cell, but for simplicity, we consider all elements to be stacks of arbitrary size. Based on these state vector characteristics, the allocation and deallocation operations are defined in the following sections.

### 4.3.3 Modeling Dynamic Variable Allocation

Execution of the simple statement **new**(P) allocates a new identified variable and assigns P to point to it. This action is modeled by (a) changing n elements from inactive to active, where n is the number of elements needed for the newly created structure, and (b) assigning the element associated with P the address of the first element of the allocated structure. Each element that is made active in this step is assigned the 4-tuple

$$( \text{ivalue}_i, \text{UNDEFINED}, \text{state}, a_i).$$

In addition, the 4-tuple

$$( a_f, \text{stmt}, \text{state}, b )$$

is assigned to pointer P. In the first 4-tuple, $a_i$ is the system allocated address for element i, and in the second 4-tuple, $a_f$ is the address of the first newly allocated element. This address is found by

$$a_f = \text{Address}(S, v), \text{ where } v = \text{NumActive}(S) - n + 1,$$

which extracts the address of the first newly activated element from state vector S. Finally, the last element, b, of the second 4-tuple represents the address of the pointer P.

The state transition function ALLOC performs the state modifications necessary to model the affect of an allocation statement. It is defined as follows.

---

**ALLOC**: STATES → STATES

**ALLOC** ≡ $\lambda$S.REPLACE"(S, NS, 1, (next(S), $\text{top}(S_{NS})|_1$, $\text{top}(S_{NS})|_3 + 1$, UNDEFINED ) )

o $\lambda$S.REPLACE"(S, pos(S), 1, (addr(S), $\text{top}(S_{NS})|_1$, $\text{top}(S_{NS})|_3 + 1$, $\text{top}(S_{\text{pos}(S)})|_4$ ) )

o $\lambda$S.ADD(S, size(S))

---

This definition of ALLOC uses the functions size(S), addr(S), and pos(S) which extract needed information from S, the current state. Knowing that Contents(S, NS) points to the statement **new(P)**, these three functions are defined as follows.

- Size(S) determines the number of elements that are allocated by the statement. Effectively, this is the size of the record (in terms of structure elements) pointed to by P.
- Addr(S) extracts the address of the newly allocated element, that is the address at which P will point. Formally,

  addr(S) = Address(S, v), where v = NumActive(S)-size(S) + 1.
- Pos(S) returns the position in state S of the pointer. Thus, in the example above, pos(S) = Index(S, P).

ALLOC also uses REPLACE", the Level 3 version of the function that modifies a value in a single cell. As a reminder, this function has 4 arguments: a state, an element number, a position in that element, and the new value. The pointer's new value

$(\text{addr}(S), \text{top}(S_{NS})|_1, \text{top}(S_{NS})|_3 + 1, \text{top}(S_{pos(S)})|_4\ )$

consists of the address of the first newly allocated element, the statement and state number of the allocation and the current address of the pointer cell. This current address is extracted from the address portion of the current top cell of the pointer's stack. To remove an identified variable, use the function DEALLOC, defined in the next section.

### 4.3.4 Modeling Dynamic Variable Deallocation

The deallocation of an n element dynamic variable is modeled by removing n state vector elements from the state vector, starting at the element whose address is the same as that stored in the pointer variable. The pointer itself is not modified. The following state transition function does the modification.

---

**DEALLOC**: STATES $\rightarrow$ STATES

**DEALLOC** $\equiv \lambda$S.REPLACE"(S, NS, 1, (next(S), top($S_{NS}$)$|_1$, top($S_{NS}$)$|_3 + 1$, UNDEFINED ) )

o $\lambda$S.DEL(S, ele(S), size(S) )

---

DEALLOC uses the functions ele(S) and size(S) which extract needed information from S, the current state. Knowing that Contents(S, NS) points to the statement **dispose**(P), then these two functions are defined as follows.

- Ele(S) returns the index of the first cell that is to be deallocated, that is, the cell at which the pointer P points. Thus

  $\mathrm{Ele}(S) = (\ \mathrm{FindAddr}(S, \mathrm{top}(S_{pos(S)})|_1)\ )|_1,$

  where pos(S), as defined in the previous subsection, returns the position of the pointer P in state S.
- Size(S) determines the number of elements that are to be deallocated by the statement. Effectively, this is the size of the record (in terms of structure elements) pointed to by P.

### 4.3.5 Modeling Level 1 and Level 2 Statements

The different state vectors and action functions used in the Level 3 model require that the definitions of the state transition functions for the Level 1 and Level 2 statements be reconsidered. Because the Level 2 functions, CALL and RETURN, use the action functions PUSH and POP, and not REPLACE′, the only change required in their definitions is using PUSH′ and POP′ (which, as defined in 3.6.2, operate on stacks of 4-tuples rather that 3-tuples) in place of PUSH and POP. These definitions are not repeated here.

The Level 1 functions are redefined in section 4.2.7 to account for the stacks in the state vectors. Extending that redefinition to use REPLACE″ instead of REPLACE′ accommodates the Level 1 state transition functions to the Level 3 model by allowing the replacement of cells that are not on the top of an element stack. Remember that at Levels 2 and 3, elements are stacks of cells and the top cell of a stack is in position one. The new definitions of the Level 1 functions are given below, without further comment.

**ASSIGN**: STATES $\rightarrow$ STATES

**ASSIGN** $\equiv \lambda$S.REPLACE"(S, NS, 1, (next(S), top($S_{NS}$)$|_1$, top($S_{NS}$)$|_3 + 1$, UNDEFINED ) )

o $\lambda$S.REPLACE"(S, iden(S), 1, (expr(S), top($S_{NS}$)$|_1$, top($S_{NS}$)$|_3 + 1$, top($S_{iden(S)}$)$|_4$))

where

iden(S) returns the index of the element of S that corresponds the identifier and expr(S) returns the value of the expression evaluated under state S.

**WHILE**: STATES $\rightarrow$ STATES

**WHILE** $\equiv \lambda$S.REPLACE"(S, NS, 1,

( cond( bvalue(S), td(S), fd(S) ) , top($S_{NS}$)$|_1$, top($S_{NS}$)$|_3 + 1$, UNDEFINED ) )

where bvalue(S) returns the boolean value of the test of the while statement, and cond(a, b, c) returns b or c if a is true or false, respectively.

**IF**: STATES $\rightarrow$ STATES

**IF** $\equiv \lambda$S.REPLACE"(S, NS, 1,

( cond(bvalue(S), td(S), fd(S) ), top($S_{NS}$)$|_1$, top($S_{NS}$)$|_3 + 1$ ), UNDEFINED )

where bvalue(S) returns the boolean value of the test of the if statement, and cond(a, b, c) returns b or c if a is true or false, respectively.

This completes the definition of the model for Level 1, Level 2 and Level 3 languages. The final section of this chapter describes the execution histories that represent program execution.

## 4.4 Execution Histories

A computation is modeled as a possibly infinite sequence of state vectors

$S^0, S^1, S^2, \ldots$

We sometimes refer to state $S^I$ as state I, meaning the state whose sequence number is I. When a state's sequence number is not relevant to a discussion, we simply use "S" to refer to an arbitrary state vector in an execution history. This introduces no ambiguity because for each state $S^I$ in an EH, State(S, NS) gives the sequence number of state S and thus State($S^I$, NS) = I.

As mentioned in the example given in section 3.3, the initial state $S^0$ is initialized so that element NS points to the program statement that is to be executed first, and the other elements are given system defined initial values. To be more explicit, we can define the initial 4-tuple assigned to an element from each of the four groups defined in subsection 4.3.2. Element NS (group 1) is assigned

(FIRST, UNDEFINED, 0, UNDEFINED)

where FIRST is the sequence number of the first program statement to execute, the statement at which this tuple is assigned is undefined, the current state is 0, and the address of element NS has no meaning so its fourth component takes the value UNDEFINED. For elements in the second group, each global variable is assigned the 4-tuple

($IVALUE_i$, UNDEFINED, 0, $ADDR_i$)

where $IVALUE_i$ is the value initially found in address $ADDR_i$, the memory location associated with the element i; and the second and third components are defined as they are for element NS. Elements that are allocated to procedure parameters and local variables are all initialized to newstk, i.e. to empty stacks. Finally, state $S^0$ has no identified variables and so the last group contains no elements.

Now, for each $i \geq 0$,

$S^{i+1} = F(S^i)$,

where F is the state transition function that models the statement to which element Contents($S^i$, NS) points. When $S_{NS}$ takes the value UNDEFINED, the computation terminates.

### 4.4.1 Execution History Classification

As discussed in the next chapter, the error detection and fault localization techniques that are applied to an execution history are based on that history's termination status. Partitioning execution histories by their status therefore gives a classification scheme that can guide the automatic debugging process. Based on this guideline, Figure 7 shows that finite state vector sequences may be abnormally terminated, Class A, or normally terminated, Class C, and that infinite sequences fall into Class B, that of nonterminated histories. In this section we discuss methods for placing an EH into its appropriate class.

### 4.4.2 Class A Execution Histories

A Class A execution history describes a computation that halts abnormally when the processing environment is unable to continue the computation because some program statement violates a standard of the language. The conditions that can halt

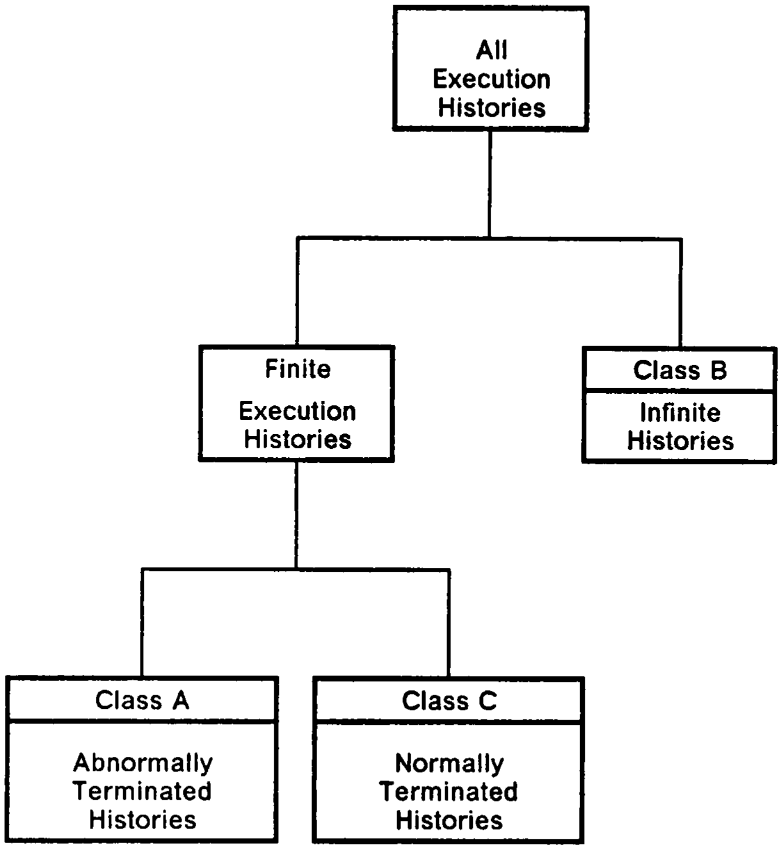

All Execution Histories
Finite Execution Histories
Class B
Infinite Histories
Class A
Abnormally Terminated Histories
Class C
Normally Terminated Histories

**Figure 7. Execution History Classification.**

the processor are described in the next chapter. Because a computation's termination status is available from the processing environment, an execution history can easily be classified as Class A. By assumption, the cause of the termination is not available from the processing environment, and therefore we use an error substantiation algorithm (described also in the next chapter) to locate the exact condition that is violated and the expression and value that cause the violation. This information about the error is used to guide further debugging analysis. Future work will incorporate the assumption that the processor provides information about the type of error that caused termination.

### 4.4.3 Class B Execution Histories

A Class B execution history is never terminated, and thus the processing environment can only give a termination status of unterminated for an EH of this class. However, an infinite state vector sequence in which each state vector is effectively finite must have two states that match (in the sense described below) because there can be only a finite number of unique state vectors. Identical state vectors provide identical initial conditions for the execution of some statement at different points of the computation, and consequently the computation will proceed from the first state with the same state vector values as from the second state. This can only occur if the program is in an infinite loop. We can therefore say that a match of state vectors is a necessary and sufficient condition for the existence of an infinite loop. Further, a state match in an execution history implies that some statement is executed twice with identical initial conditions. This implies that the sequence of states that follows the first matching state is identical to the sequence following the second. The second

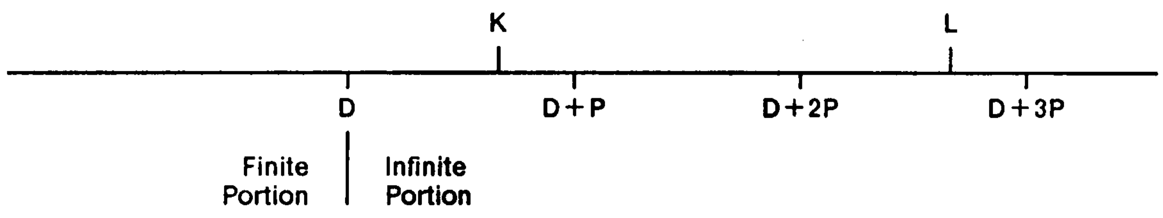


**Figure 8. A Class B Execution History:** The State Match Algorithm detects a matching pair of states (K and L). The point D divides the EH into a finite and infinite portion. The infinite portion repeats the states D to D + P.

state is a member of the sequence following the first state, and consequently that sequence consists of a cycle of states that repeats. A first repetition of this cycle must exist. Thus if P is the length of the cycle and D is the index of the first state of the first cycle, i.e. the first state of the infinite loop, then $S^N$ and $S^{N+P}$ match for $N \geq D$. The first state of this first cycle (not necessarily the first state of the loop) divides the sequence into a finite part preceding the first state in the repetition and an infinite part following that state (see point D in Figure 8). This infinite EH therefore has a finite representation consisting of the finite part of the EH plus the finite cycle that repeats in making up the infinite part of the EH.

Based on these observations, the following algorithm will detect a matching pair of states, K and L in Figure 8, and thus the presence of an infinite loop and the existence of a Class B Execution History.

**The State Match Algorithm**

```
I := 0
while $S^I$ and $S^{2*I}$ do not match
    I := I + 1
```

Two states are said to match if they have the same number of active elements and if respective active elements are identical (except for their third components). Because the state vectors for these languages are allowed to be extended with newly allocated cells, the conclusion that a matching pair of states must exist would appear to not hold for Level 3 languages. However, because each newly allocated cell must be assigned an address, the number of states that differ in their active elements is still finite and thus the conclusion still holds.

The following argument guarantees that the State Match Algorithm terminates and is linear. As discussed above, there exist D and P such that $S^N$ and $S^{N+P}$ match for $N \geq D$, and therefore, if $I \geq D$ and P divides I then $S^I$ and $S^{2*I}$ match. Now the integers between D and D + P, inclusive, must contain an I that is divisible by P, and therefore the State Match Algorithm terminates and the number of comparisons needed to find a pair of matching states is linear in D and P. Of course, as noted by Minsky [MINM67], the number of possible distinct states that a machine can assume before repeating is far larger than can practically be checked, however, because the state contains values of only variables that are used, this number can be much smaller and a repetition may occur much earlier. Finally, we note that this algorithm is essentially the same as the algorithm, attributed by Knuth [KNUD81] to R.W. Floyd, for finding a cycle in the sequence of integers $X_0$, $X_1$, ..., where $0 \leq X_i \leq m$ and $X_i = f(X_{i-1})$, for some function f for which $0 \leq f(x) \leq m$ if and only if $0 \leq x \leq m$.

### 4.4.4 Class C Execution Histories

Class C execution histories terminate normally and are therefore easily classified by obtaining the computation's termination status from the processing environment. However, such a computation is not necessarily correct: it may contain errors that are not detected by the processing environment, and its output may be incorrect. Therefore the error classification scheme of the next chapter includes errors that are not detected by the processing environment and that do not affect termination. We are not concerned with computations that produce invalid output as are some [SHAE82, MILB88]; a combination of their techniques with ours is a topic of future work.

This concludes the description of execution histories. The next chapter describes the errors that can occur within an EH, and Chapter 6 gives an algorithm for automatic debugging that is based on this analysis of errors and execution histories.

# 5.0 Errors

This chapter discusses five aspects of errors:

1. Classification,
2. Characterization,
3. Detection,
4. Substantiation, and the
5. Relationship between error types and execution history classes.

The classification scheme divides errors into three *types* based on their effect on termination. The second aspect, characterization, defines distinguishing error characteristics, while detection uses these characteristics to locate errors within an EH. After detection, substantiation determines the details of the particular instance of an error. The final aspect describes the relationship between the error types and the execution history classes. This relationship guides the automatic debugging strategy explained in the next chapter.

To clarify this research, the following section defines our use of the term *error*. After this definition, Section 5.2 describes the error classification scheme; and, because characterization, detection, and substantiation are closely related, the following section describes these three aspects together. The final section of this chapter describes the error type and execution history class relationship.

## 5.1 Error Definition

In this discussion, an *error* is defined to be program *behavior* that violates any of the following requirements for correct program behavior:

- the program should terminate,
- it should adhere to all requirements of the standard of the language, and
- it should avoid certain conditions called *anomalies* that, while not violating any requirements of the language standard, indicate the possibility of incorrect program logic.

Based on these criteria, program errors include

- infinite loops,
- language standard violations such as division by zero, assignment of an incompatible value, and dereferencing a nil pointer, and
- anomalies such as assigning a variable a value at two locations without an intervening reference to that variable's value.

This class of errors would be broadened if the criteria for correct behavior were strengthened by including a requirement that an individual program should meet the specifications defined for that program. However, because the debugging system is assumed to operate without any knowledge of a program's intended behavior, we do not include such a requirement and thus consider a smaller class of errors. Specifically, based on the criteria given, incorrect output from a program is not defined to be an error.

Underlying errors are source code conditions called *faults*. As an example, a fault such as modifying a loop variable outside rather than inside a loop could underlie an infinite loop error. The process of detecting the faults underlying the errors detected in an EH is the subject of the following chapter of this dissertation.

Our definition of an error is derived from the IEEE Pascal Standard [IEEE83a] which defines an error to be a standards violation that, for some programs, can only be detected by program execution and not by a static program analysis (i.e. by the compiler). Division by zero, for example, is an error by this definition, because in some programs the division by zero violation can only be detected by running the program. Our class of errors encompasses this use of the term error and extends it by adding the conditions that the program should terminate and avoid anomalies. This usage differs from the *IEEE Standard Glossary of Software Engineering Terminology* [IEEE83b] in which an error is defined to be "human activity resulting in software containing a fault," a fault is defined to be a condition "in software [that] may cause a failure if encountered," and a failure to be "a departure of program operation from program requirements." This definition of a fault corresponds closely to our own, while our concept of an error is closer to the Glossary's definition of a failure. However, rather than adopting these definitions, we maintain the intuitive notion of an error as incorrect program behavior and use the definitions given above for errors and failures. Using this concept of an error, the next section of this chapter describes an error classification scheme.

## 5.2 Error Classification

Based on their effect on program termination, errors can be divided into the following three categories.

- The Type 1 category contains errors that cause immediate termination,
- the Type 2 category contains errors that prevent termination, and
- the Type 3 category contains errors that do not effect termination.

For simplicity, we call an error in the Type 1 category a Type 1 error and similarly refer to Type 2 and Type 3 errors. Each Type 1 error involves a standards violation that, by assumption, is detected by the processor and causes termination. As shown in Figure 9, these violations are divided into value, pointer and availability errors. The *value errors* are expressions whose values are invalid, either in their specific context, e.g. sqrt(-abs(x)), or in any context, e.g. MAXINT+1. A *pointer error* occurs when a dangling or NIL pointer is used; an *availability error* occurs when an allocation statement or a call of a user's procedure cannot execute because there is not enough available memory to allocate either the identified variable or the parameters of the procedure. Based on these definitions, Figure 10 details specific value, pointer and availability errors. In contrast to the Type 1 errors which cause immediate termination, the Type 2 errors prevent termination. Thus, this category contains the in-

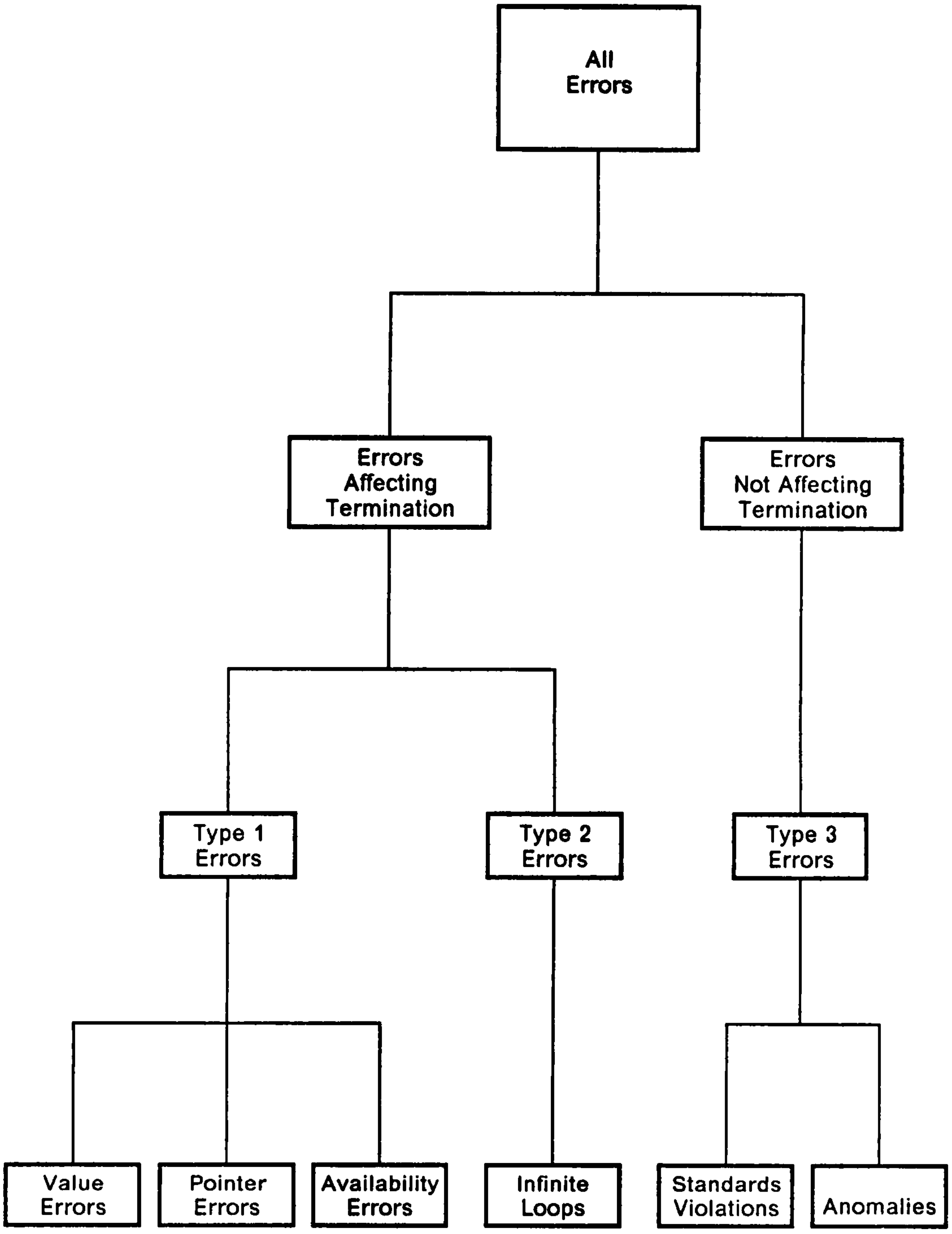


**Figure 9. Error Types**

- Type 1 Errors
  - Value errors
    - Array subscript not assignment compatible
    - Invalid parameter for system functions: sqr, ln, sqrt, trunc, round, chr, succ, and pred
    - DIV, MOD, or / by 0
    - Overflow or underflow
  - Pointer errors
    - Use (dispose or dereference) of a dangling reference
    - Use (dispose or dereference) of a pointer whose value is NIL
  - Availability errors
    - Insufficient memory available for allocation of an identified variable, and
    - Insufficient memory available for allocation of the arguments of a procedure.
- Type 2 Errors
  - The occurrence of an infinite loop within an EH
- Type 3 Errors
  - Standards Violations
    - Use of the value of an element that is
      - incompatible,
      - corrupt, or
      - undefined,

      but that is not a dangling reference.
    - Generation of an element that is
      - incompatible or
      - corrupt.
  - Anomalies
    - The existence of garbage in a state vector
    - The existence of dangling references in a state vector
    - Dynamic define-define (DD) errors
    - Dynamic define-undefine (DU) errors

**Figure 10. Outline Listing of Errors**

finite loops. The final error category, Type 3, includes anomalies and those standards violations that are not detected by the processing environment. As outlined in Figure 10, these particular standards violations include the generation or use of an incompatible or corrupt element, and, in addition, the use of an undefined element. As defined in the previous chapter, an element is incompatible if its value is not valid for the type of its associated variable and is corrupt if its value is assigned by an assignment statement that uses an incompatible pointer to access the element. An undefined element is one that has not yet explicitly been assigned a value by a program statement. The second Type 3 group, the anomalies, include garbage elements and dangling references in a state vector, and dynamic define-define and define-undefine errors. The latter two errors occur when two states define (or define and undefine) a variable without an intervening reference to the variable. Each of these errors is discussed in more detail in the following section.

## 5.3 Error Characterization, Detection and Substantiation

An error's effect on termination reveals significant characteristics about that error which directly influence the techniques used in its detection and substantiation. Therefore, this section describes error characterization, detection and substantiation with respect to each of the three error types.

### 5.3.1 Type 1 Errors

A Type 1 error is characterized by (1) an abnormally terminated execution history and (2) the specific condition whose violation halted the computation. The first characteristic, which can be detected by examining the computation's termination status, indicates that an unspecified Type 1 error has occurred; determining the condition violated, and thus the exact error that has occurred, requires additional information. Making no assumption about information that is available from the processing environment, the algorithm Type_1_Substantiation (see Figure 11) determines the exact error that has occurred by evaluating the statement that failed and reproducing the error that caused termination. To do this, it examines the operations of the erroneous statement in the order followed by the processing environment. If each operation meets certain conditions that guarantee that it can execute successfully, then it is evaluated and its value is stored as an intermediate value for an operand that is evaluated later. If the operation does not meet the conditions, then the operation and the condition that is violated are assumed to reflect the error that caused termination.

In the algorithm shown in Figure 11, the operations of the erroneous statement are stored in OPRNS, and INTVALS holds the intermediate values of the respective operations. The main loop of this algorithm (beginning with statement 1) processes each operation, $\phi$, in turn and uses the function MeetCond to verify that $\phi$'s operands in OPRNDS meet each of the conditions found in CONDS (see statement 2). We do not define MeetCond further except to note that it uses INTVALS and state $S^T$ (i.e. the terminal state of the EH) to determine the values of the operands of $\phi$.

If MeetCond determines that one of the operation's conditions does not hold, then an error has been detected, and in statement 3 algorithm

```
(* Algorithm Type_1_Substantiation *)

(*  ARRAYS
    OPRNS[1..M] holds an ordered list of operations
    INTVALS[1..M] holds the intermediate values of the operations in OPRNS
    OPRNDS[1..N] holds the operands of an operation
    CONDS[1..P] holds the conditions that must hold for a successful operation *)

T := the number of the last state of the EH
OPRNS[1..M] := the M operations of the statement that terminated, in the order
                      that they are executed by the processing environment

ERRSUB := false  (* becomes true if the error is substantiated *)

For I := 1 to M (* I is the number of the operation being evaluated *)          (* 1 *)

    $\phi$ := OPRNS(I)        (* $\phi$ is the operation being evaluated *)
    OPRNDS[1..N] := the N operands of operation $\phi$
    CONDS[1..P] := the P conditions that must hold for $\phi$ to be applied correctly

    For J := 1 to P                                                              (* 2 *)
        ERRSUB := not MeetCond(CONDS[J], OPRNDS, INTVALS, $S^T$ )
    until ERRSUB

    If ERRSUB
        then Announce_Type_1_Information                                         (* 3 *)
        else INTVALS[I] := the value of $\phi$ evaluated under state $S^T$ and INTVALS      (* 4 *)

until ERRSUB

end  (*  end of Type_1_Substantiation  *)
```

**Figure 11. Algorithm Type_1_Substantiation**

```
(* Algorithm Announce_Type_1_Information *)

    announce a Type 1 error occurred
    announce state T is the state of interest

    case $\phi$ of
       dereference:
          announce a pointer error occurred
          announce the variables in the pointer expression are of interest
          announce the condition violated is CONDS[J]

       call of the allocation routine or a user defined routine:
          announce an availability error occurred
          announce the variables used in the call are of interest
          announce the condition violated is insufficient memory

       else announce a value error occurred
          announce the variables of interest are in the set
              {v| v is a simple variable for which v = OPRNDS(k) for some $1 \le k \le N$}

          announce the condition violated is CONDS[J]

     end (* case *)

end (* Algorithm Announce_Type_1_Information *)
```

**Figure 12. Algorithm Announce_Type_1_Information**

Announce_Type_1_Information (shown in Figure 12) determines information about that error and announces that information to the debugging system. Once this information has been announced, the loop completes and (with one final test) the algorithm terminates. However, if MeetCond returns true for each of the P conditions in CONDS, then $\phi$ is assumed to have executed correctly in the program, and so it is evaluated and the resulting value is stored in INTVALS (see statement 4).

Because the statement under evaluation is known to contain an error, the substantiation algorithm is guaranteed to detect a violated condition. When this occurs, the algorithm calls Announce_Type_1_Information (see Figure 12) to announce information about the error that caused termination. Based on the operation whose condition is found to be invalid, this algorithm announces the following information to the debugging system:

- that a Type 1 error has occurred,
- the state at which the error has occurred,
- the specific error that has occurred,
- the variables involved in the error, and
- the condition that has been violated.

Relative to a particular operation $\phi$, the conditions that must hold for $\phi$ to operate correctly are shown in Figure 13. Most of these conditions are self-explanatory, but the second condition for both dereference and dispose requires additional comment. In these two conditions, FindAddr(S, P) attempts to locate the cell to which P points (i.e. the cell that has the address found in pointer P) and to return that cell's element number and stack position if the cell exists, and to return (UNDEFINED, UNDEFINED) otherwise. Thus each of these conditions states that a pointer which is used can not

| OPERATION | CONDITION(S) |
|---|---|
| unary - {integer} | $X \neq$ MININT |
| sqr {integer} | sqr(X) $\leq$ MAXINT |
| sqr {real} | sqr(X) $\in$ REAL, the set of representable real numbers |
| sqrt {real} | $X \geq 0$ |
| ln {real} | $X > 0$ |
| pred | pred(X) exists |
| succ | succ(X) exists |
| +, -, * {integer} | MININT $\leq$ X op Y $\leq$ MAXINT |
| +, -, * {real} | X op Y $\in$ REAL |
| / | X/Y $\in$ REAL |
| | $Y \neq 0$ |
| X div Y | $Y \neq 0$ |
| X mod Y | $Y > 0$ |
| X[Y] (subscript) | Y within X's declared index bounds |
| dereference | $P \neq$ nil |
| | FindAddr(S, P) $\neq$ (UNDEFINED, UNDEFINED) |
| new(P) | \|Addresses\| - \|{a\|a is an address in an active cell}\| is greater than the number of cells to be allocated |
| dispose(P) | $P \neq$ nil |
| | FindAddr(S, P) $\neq$ (UNDEFINED, UNDEFINED) |
| procedure(X,...,Y) | \|Addresses\| - \|{a\|a is an address in an active cell}\| is greater than the number of cells to be allocated by the call |

**Figure 13. Conditions for Type 1 Error Substantiation:** To execute correctly, each operation must meet the condition(s) shown.

be a dangling reference. Note that with the exception of the operations of array subscripting, dispose, and dereferencing, the operations do not require their operands to be of the correct type. Instead they treat their operands as correct, regardless of type. For example, because the boolean operations simply operate on the low bits of their arguments and do not terminate if given an argument of the wrong type, there are no conditions on their arguments and they do not appear in Figure 13.

### 5.3.2 Type 2 Errors

Type 2 errors, i.e. errors that prevent termination, are partly characterized by an EH that contains a matching pair of states. As discussed in subsection 4.4.3 of the previous chapter, the State Match Algorithm locates a matching pair of state vectors in a Class B execution history and thus detects the presence of a Type 2 error. Using this pair of states, the substantiation algorithm (see Figure 14) detects and calculates the following characteristics of an infinite loop error:

- the length of a cycle of the EH (i.e. LEN),
- the first cycle of the infinite loop, delineated by States D and E,
- the program statement sequence numbers of the first and last statements of the infinite program loop (i.e. statements FIRST and LAST), and
- the state numbers of the first cycle that begins with the first statement (FIRST) of the loop (i.e. states START and FINISH).

Beginning with the pair of identical states that were detected by the State Match Algorithm, states K and L in Figure 15, section (* 1 *) of the substantiation algorithm finds the cycle length by finding the first state $S^I$ such that $S^I$ matches $S^K$ and $I > K$.

```
(* Algorithm Type_2_Substantiation *)

Get K and L              (* The matching states located by the State Match Algorithm *)

                    (* Find Length of Cycle *)
I := K
Repeat I := I + 1                                                          (* 1 *)
   Until $S^I$ and $S^K$ match
LEN := I-K

                 (* Find the first cycle [$S^D$ .. $S^E$] *)
I := K
Repeat I := I - 1                                                          (* 2 *)
   Until $S^I$ and $S^{I+LEN}$ do not match
D := I + 1
E := D + LEN

             (* Find FIRST and LAST, the sequence numbers of the
                first and last statements in the loop      *)
TRUECOND  := {}
FALSECOND := {}
For I := D to E                                                            (* 3 *)
   If the statement pointed to by Contents($S^I$, NS) contains a loop condition
      then if the condition's test is true
         then add Contents($S^I$, NS) to TRUECOND
         else add Contents($S^I$, NS) to FALSECOND
FIRST := the member of the singleton TRUECOND - FALSECOND
LAST  := the sequence number of the last operation of the loop that starts with FIRST

              (* Find START and FINISH *)
I := D
While Contents($S^I$, NS) $\neq$ FIRST do                                  (* 4 *)
   I := I + 1

START := I
FINISH := START + LEN

announce the infinite loop error                                           (* 5 *)
announce D, the dividing point
announce statements FIRST and LAST
announce states START and FINISH
announce the variables of the loop's condition are of interest

End  (* of Type_2_Substantiation *)
```

**Figure 14. Algorithm Type_2_Substantiation**

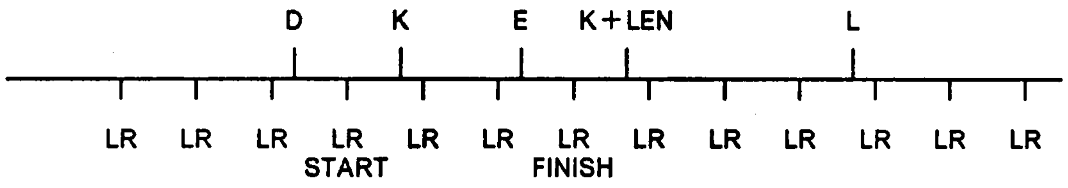


**Figure 15. An Execution History Illustrating Type 2 Substantiation:** The Type 2 Substantiation algorithm detects LEN, the cycle length, [D..E], the first infinite cycle in the EH, and [START..FINISH] the first cycle that begins with a repetition of the loop condition. Each point LR (Loop Repetition) marks the beginning of a repetition of the loop.

This gives LEN := I-K. Section (* 2 *) of the algorithm then finds the first cycle in the execution history by finding the first pair of states $S^I$ and $S^{I+LEN}$ that match but whose respective predecessors do not match. This first cycle begins and ends at points D and E, respectively. Moreover, State D divides the EH into a finite and an infinite portion, and D is the first state of the infinite portion. Substantiation's next task, in section (* 3 *), is to delineate the statements that comprise the loop that does not terminate. To do this it locates FIRST, the number of the statement (within the infinite loop) whose condition, when evaluated, always indicates "loop again". Only one such condition can exist because any other loop condition within the infinite loop must, at some point in the computation, have evaluated to a terminate condition. Once FIRST is found, the last statement of that loop which begins with statement FIRST is found and its sequence number is assigned to LAST. Thus the statements delineated by and including FIRST and LAST comprise the infinite loop. Next, in section (* 4 *), substantiation searches forward from state D to find [START..FINISH], the cycle in the EH that begins with the first execution of statement FIRST, the loop's condition, and finally, in section (* 5 *), it announces relevant error information to the debugging system and terminates.

### 5.3.3 Type 3 Errors

This subsection describes the characterization, classification and substantiation of Type 3 errors, that is, of the errors that do not affect program termination. Standards violations, the more complex of the Type 3 errors, are considered in subsection 5.3.3.1 below. The remaining Type 3 errors, the anomalies, are discussed in subsection 5.3.3.2.

### 5.3.3.1 *Standards Violations*

As defined in Section 5.2, the Type 3 standards violations include

- the use of
  - undefined,
  - incompatible, and
  - corrupt elements; and
- the generation of
  - incompatible and
  - corrupt elements.

To simplify the discussion of these errors we introduce functions that map a variable to the state vector element associated with that variable, that locate the allocation point of an identified variable, and that determine if an element is undefined, incompatible or corrupt in a given state vector. These functions are the subject of the subsection below. Following subsections describe in turn the characterization, detection and substantiation of the Type 3 standards violations.

**Function Definitions:** This subsection defines five functions that are used both in the discussion of Type 3 standards violations and in the automatic debugging algorithms presented in the next chapter. Within the scope of this discussion, we only provide intuitive definitions of the functions, followed by an explanatory discussion of each.

| | |
|---|---|
| **Index(S, V)** | returns the number of the element that is associated with the variable identifier (defined below) V in state vector S. |

| | |
|---|---|
| **FindAlloc(S, e)** | returns the number of the state that first reflects the allocation of the identified variable which corresponds to element $S_e$. |
| **IsUndefined(S, e)** | returns true if element number e is undefined in state vector S and false otherwise. |
| **IsIncompatible(S, e)** | returns true if element number e is incompatible in state vector S and false otherwise. |
| **IsCorrupt(S, e)** | returns true if element number e is corrupt in state vector S and false otherwise. |

As defined in subsection 4.3.1, an identified variable is a variable, usually a record, that is dynamically allocated by a **new** statement. The terms incompatible and corrupt are also informally defined in that subsection; detailed characteristics of incompatible and corrupt elements are given in the discussion that follows. An undefined element is one that has not been assigned a value by a program statement.

The first of these functions, Index(S, V), returns the number of the element that is associated with the variable identifier V. A variable identifier is either the name of a simple variable, e.g. X; an array expression, e.g. ARR[I]; or a pointer expression, e.g. P↑.NEXT. If variable identifier V denotes a global variable, an identified variable, a local variable, or a value parameter of a procedure, but not a pass by reference formal parameter, then the number of the element that corresponds to V can be determined by a static analysis of the source program, supplemented as needed by values from state vector S. We assume without further discussion that such information is available, as needed, to the debugging system. This static information is sufficient to determine Index(S, V) if V is one of the variable identifiers described above, but if V is a pass-by-reference, formal parameter, then additional information is needed because the element associated with a reference parameter can change during program execution. Specifically, if V is a pass-by-reference formal parameter of procedure P and if P has been called but not exited in state S, then for the duration of that call, the element that is associated with V is the element that is associated with V's corresponding actual parameter. To find this actual parameter, Index searches the EH backwards from state S and finds the state that precedes the call statement which invoked the procedure. Index then uses this call statement and state to determine the actual parameter. It can then return the number of the element that corresponds to this actual parameter as the number of the formal parameter. Thus, Index(S, V) can calculate the number of the element that is associated with any variable identifier V.

The purpose of the next function, FindAlloc, is to locate the state at which an identified variable is allocated. Thus, for element number e of state vector S for which element e corresponds to an identified variable, FindAlloc(S, e) returns the number of the state that results from the allocation of the identified variable associated with element e. To do this, FindAlloc relies on two facts about newly allocated elements. First, such an element has not been assigned a value by a program statement and thus has UNDEFINED as its second component, and second, the number of the state at which allocation has occurred is the undefined element's third component. Therefore, to find the allocation of element number e, FindAlloc first examines each change in element e's value in order to find the transition that changes e's statement component from UNDEFINED to a specific value; FindAlloc then returns the third component of the initial state of this transition. The code to perform this search follows.

```
Function FindAlloc(S, e)
 I := State(S, NS)  (* I is the sequence number of the current state S *)
    (* The allocation statement leaves the newly allocated cell undefined.
     To find the state at which this allocation occurs, FindAlloc examines
     the state that precedes each assignment of element e
     to see if element e is undefined in that preceding state *)
 While not IsUndefined($S^I$, e) do
   I := State($S^I$, e) - 1
    (* When you get here, element $S^I_e$ has just been created. *)
    (* Return the number of the state where allocation occurs *)
 Return State($S^I$, e)
```

Within this code, variable I initially takes on the value of the sequence number of state S and subsequently (i.e. within the **while** loop) takes the value of the sequence number of each state that precedes successively earlier assignments of element $S_e$. To calculate these values, State($S^I$, e) gives the number of the state reflecting an assignment of $S_e$, and this value minus one gives the number of the state preceding the assignment. When $S_e$ has the value UNDEFINED in the state preceding an assignment then that assignment must be the first assignment to $S_e$ after the allocation. Therefore when the **while** loop has completed, State($S^I$, e) provides the sequence number of the state that immediately follows the allocation of the identified variable associated with element number e, and FindAlloc returns this value and terminates. Note that this backward search of the EH is *not* sequential, but follows a sequence of value assignments to the allocation of the identified variable denoted by e.

The next three functions, IsUndefined, IsIncompatible, and IsCorrupt are used to query the status of a particular element of a state vector. For Level 1 languages these functions are defined on all elements of a state vector. For Level 2 and Level 3 languages they are defined only on those elements that are non-empty stacks in the given state vector, that is, only on elements that correspond to global variables, to identified variables, or to parameters or local variables of procedures that have a current activation represented in the state vector. Thus, the functions are not defined on elements that correspond to parameters or local variables of procedures which do not have a current activation, because these elements exist in the state as empty stacks (see subsection 4.2.2). In addition, for Level 3 languages, the functions are only defined on the active elements (see subsection 3.6.1.5) of the infinite vector. As used in this chapter and the next, these functions are applied only to elements that correspond to variables that are actually accessible in the context of the current state, and thus there is no danger of applying them to elements on which they are not defined. Having considered the common domain of these three functions, we can now define each in turn.

Before any accessible variable (i.e. one whose associated element is not an empty stack) is assigned a value by a program statement, its associated element has UNDEFINED as its second component; after an assignment the second element has the number of the statement that performed the assignment. Thus if e is the number of an arbitrary accessible element of an arbitrary state vector S, then

$$\text{IsUndefined}(S, e) \equiv (\text{Stmt}(S, e) = \text{UNDEFINED}).$$

Note that by this definition, an element that has not been assigned a value but that has some system defined initial value is considered to be undefined. This corresponds to our intuitive use of the term and allows us to model both the error of using a variable before it has been assigned a value and also errors that result from the use of an inappropriate system defined initial value.

The second function, IsIncompatible(S, e), returns true if the value of element number e in state vector S is not within the type of the variable associated with that element. For most elements the variable associated with an element is fixed and can be determined by a static program analysis, but for elements that are associated with identified variables (i.e. with dynamically allocated variables) this association can change as the program executes, and thus an EH analysis is needed to discover the type of the associated variable. Using the function FindAlloc, IsIncompatible can locate the statement at which allocation occurred, and from this statement it can determine the associated variable. Once this variable is determined, either using FindAlloc or by a static program analysis, the type of the variable and thus the type associated with the element can easily be found from the variable's declaration. Determination of a variable's type and of whether a value is within a type are both outside the scope of this dissertation and are not considered further here.

Considering now the final function, IsCorrupt, recall, as defined in subsection 4.3.1, that an element of a state vector is corrupt if and only if its value is assigned by an assignment statement whose left side uses an incompatible pointer to access the variable associated with the element. For an arbitrary element e in state vector S, let n be the number of the statement that assigns e, that is, n = Stmt(S, e). Also let K be the number of the state that precedes the assignment of e in S, that is K = State(S, e) - 1. Using these definitions, we can see that IsCorrupt(S, e) returns true if and only if

- statement n is an assignment,
- statement n uses pointer expression P on the left side of the assignment, and
- IsIncompatible($S^K$, Index($S^K$, P)), that is, pointer P is incompatible in the state that precedes the assignment of element e.

Thus a syntactic program analysis, augmented by an execution history analysis using function IsIncompatible, effectively computes IsCorrupt(S, e) for arbitrary non-empty element e in state S.

This completes the definition of the functions that are needed to define the Type 3 standards violations. The following subsection describes the characterization of these errors.

**Characterization:** Using the functions defined in the preceding subsection, this subsection defines the distinguishing characteristics of the Type 3 standards violations. These errors include (1) the use of undefined, incompatible and corrupt elements, and (2) the generation of incompatible and corrupt elements. The first error, use of an undefined element, is characterized by a state that precedes a statement which uses an element that is undefined in the given state. More formally, a use of an undefined element occurs at state vector S if there is an element e such that

- e = Index(S, V), for some variable identifier V used in statement Contents(S, NS) and
- IsUndefined(S, e).

The first condition indicates that the variable identifier associated with element e is used in the statement following state S, and the second indicates that element e is undefined in state vector S. As an illustration of this characterization, consider the following fragment of an execution history.

| Statement | State Vector i | | | |
|---|---|---|---|---|
| | | Element 1 | Element 2 | Element 3 |
| | i | NS | A | B |
| | 100: | <(25,X,100), | (X,X,X), | (10,U,0),...> |
| 25: A := B + 1 | | | | |
| | 101: | <(X,25,101), | (11,25,101), | (10,U,0),...> |

In this example, and in the examples below, X denotes a value that is not relevant to the example, and U denotes the value UNDEFINED. In state 100, element 3 is associated with variable B which is used in statement 25, that is, 3 = Index($S^{100}$, B) and B is used in statement Contents($S^{100}$, NS). Further, IsUndefined( $S^{100}$, 3) and therefore a use of an undefined element (i.e. element 3) exists at state vector 100. This completes the characterization of the use of an undefined element. The characteristics of the use of incompatible and corrupt elements are defined analogously, with the only difference being the replacement of the function IsUndefined by either IsIncompatible or IsCorrupt. Rather than repeating these essentially identical characterizations, we move to the discussion of the generation of incompatible and corrupt elements.

An element becomes corrupt when it is assigned in an assignment statement that uses an incompatible pointer to access the element. Thus, the generation of a corrupt element is characterized by a state that follows an assignment statement which uses an incompatible pointer element on its left hand side. More formally, a generation of a corrupt element exists at state vector T if there are elements e and f such that

- State(T, f) = State(T, NS),

- statement Stmt(T, NS) is an assignment whose left side uses a pointer expression P for which e = Index(S, P), where state vector S immediately precedes state vector T, that is, State(S, NS) = State(T, NS)-1, and
- IsIncompatible(S, e).

The first of these conditions indicates that element f is assigned in the statement that precedes state T, the second indicates that this preceding statement is an assignment that uses a pointer, associated with element e, on its left side (i.e. to access element f), and the final condition indicates that element e is incompatible in state S, the state that immediately precedes state T. The following program and execution history fragment illustrates this characterization.

| Statement | State Vector i | | | |
|---|---|---|---|---|
| | | Element 1 | Element 2 | Element 3 |
| | i | NS | P | A |
| Type List = ↑Node; | | | | |
| Node = record | | | | |
| VAL: Integer; | | | | |
| next: List; end; | | | | |
| Var P: List; A: Integer; | | | | |
| | 100: | <(25,X,100,U), | (501,X,X,X), | (X,U,X,501),...> |
| 25: P↑.VAL := 10 | | | | |
| | 101: | <(X,25,101,U), | (501,X,X,X), | (10,25,101,501),...> |

Assume in this example that in state 100, element 2, associated with pointer P, points to variable A and thus execution of statement 25 makes element 3, associated with A, corrupt. In terms of the characterization, State(101, 3) = State(101, NS) implies that element 3 is modified at state 101, statement Stmt(101, NS), i.e. statement 25, is an assignment that uses pointer P and 2 = Index($S^{100}$, P), and

IsIncompatible(100, 2), i.e. element 2 is incompatible in state 100 (because 501 is not in the set of pointers to Nodes). Thus a generation of a corrupt element occurs at state 101. This example completes the characterization of this error.

Characterization of the generation of an incompatible element requires two distinct sets of characteristics because an element can become incompatible in two different ways. In the first, an element is assigned a value of the wrong type, as occurs, for example, when a variable of subrange type is assigned a value that is outside the variable's type. This error is characterized by a state vector that immediately follows a statement which assigns an element a value that is not in the type of the variable associated with the element. More formally, a generation of an incompatible value exists at state vector S if there is an element e such that

- State(S, e) = State(S, NS),
- IsIncompatible(S, e).

The first condition indicates that element e is assigned at the statement that precedes state S, and the second indicates that the element is incompatible after the assignment. If these two conditions hold, then the statement preceding state S must have given the element a value that is not within its type, and thus have made it incompatible.

An element is made incompatible by the second method when an identified variable is coincidentally allocated an address to which a dangling pointer already points. As an example, this error occurs if pointer Q is dangling before a call to new(P), and after the call Q points to one of the elements that has just been allocated. For a further example, see the code fragment shown in subsection 4.3.1 of Chapter 4. This error is characterized by a pair of successive states in which an element is dangling in the earlier state and not dangling in the latter. More formally, an incompatible element is generated in this way if there is a pair of successive state vectors, S and T (i.e. State(S,NS)=State(T,NS)-1), for which there exists an element e such that

- element e is associated with a variable of pointer type in state S,
- FindAddr(S, Contents(S, e)) = (UNDEFINED, UNDEFINED),
- FindAddr(T, Contents(S, e)) $\neq$ (UNDEFINED, UNDEFINED),
- statement Stmt(T, NS) is of the form New(P) but Index(S, P) $\neq$ e,
- State(S,NS)=State(T,NS)-1), and
- IsIncompatible(T, e).

The first two conditions indicate that element e is a dangling reference at state S. As described in subsection 3.6.1.5, FindAddr(S, ADDR) returns (UNDEFINED, UNDEFINED) only if there is no cell in S that has address ADDR. The next two conditions indicate that e is no longer dangling at state T, but that the statement new(P) that follows state S should not affect element e. The final condition indicates that after the call to New(P), element e points to a variable of the wrong type, and thus the condition rules out the rare case where element e is coincidentally and illegally caused to point to an element of the correct type. This restriction maintains the focus of the dissertation on more common errors; a complete treatment of pointer errors is the subject of future work.

This completes the characterization of Type 3 standards violations. The following discussion concerns their detection and substantiation.

**Detection:** Based on the above characterizations of the Type 3 standards violations, a simplistic scheme to detect these errors is to sequentially search an entire execution history for their distinguishing characteristics. A more sophisticated scheme is integrated into the automatic debugging algorithm given in the next chapter. In this approach, a search is made for only those errors that are related (in a sense defined in Chapter 6) to an existing Type 1 or Type 2 error. Because the detection of the Type 3 standards violations is integrated into the debugging algorithm, we leave the discussion of detection techniques for the next chapter and do not consider them here.

**Substantiation:** As in the case of error detection, the substantiation of Type 3 standards violations is also an integral part of the automatic debugging algorithms given in the next chapter. Therefore we do not give a precise substantiation algorithm here; instead we describe only the information that the substantiation process announces to the debugging environment.

For each of the Type 3 errors characterized above, substantiation announces the error that has occured and the state and statement of that occurrence. Additional information is determined and announced as needed for each specific error. For the errors involving the use of undefined, incompatible or corrupt elements, the invalid element is announced. In addition, when an incompatible or corrupt element is used, the state at which the element became invalid must also be determined and announced. If the element became invalid in an assignment statement, then the state at which that assignment occurred can be found in the third component of the element's state vector cell. However, if an element became incompatible in an allocation statement, the function FindAlloc is used to search the execution history and to find that allocation statement.

Similarly, for errors involving the generation of incompatible or corrupt elements, when an element is made incompatible by an assignment statement, then the incompatible element and the invalid value are reported. If an element is made incompatible through an allocation statement, then the pointer that is made incompatible is announced. Finally, when an element becomes corrupt, then that element and the pointer that caused the corruption are announced.

This completes the overview of the substantiation of the Type 3 standards violations. The next subsection describes the final group of errors: the Type 3 anomalies.

### *5.3.3.2 Anomalies*

To aid the discussion of the anomalies, three functions are defined. The first of these functions, **Defs(n)**, returns the set of variables that are defined (given a value) in the statement whose sequence number is n. A related function, **Refs(n)**, returns the set of variables that are referenced in statement n. As an example of the sets returned by Defs and Refs, consider the following statement.

25: P↑.NEXT↑.VAL := A + B[I]

In this example, Defs(25) = {P↑.NEXT↑.VAL} and Refs(25) = {P,P↑.NEXT,A,B[I],I}. The third function, **UnDefs(n)** returns the set of variables that are undefined in statement n. In our model, the only statement that can undefine a variable is the procedure return. It undefines the procedure's value parameters and local variables. These three functions are widely used in data flow analysis [FOSL76,OSTL81], and thus we rely on this intuitive description rather than defining them more formally.

To complete the discussion of the Type 3 errors we first consider the define-define and define-undefine anomalies, and finally we examine garbage and dangling references. The data flow anomaly define-define (define-undefine) occurs in a program if a variable is defined at two statements (defined at a statement and undefined at a later statement) and that variable is referenced at no statement between the two [FOSL76, OSTL81]. Applying these concepts to execution histories, a define-define (DD) anomaly occurs in an execution history if there is a subsequence $[S^I..S^J]$ of the EH for which there is a variable V that is in Defs(Contents($S^I$, NS)) and in Defs(Contents($S^J$,NS)) but not in Refs(Contents($S^K$, NS)) for any state K in the subsequence beginning with state I+1 and ending with state J. In other words, a DD error occurs if V is defined in the statements that respectively follow states I and J, but V is not referenced in any of the statements that follow states I+1 to J, inclusive. Note that a reference to V in the statement following state I occurs before a definition of V at that statement, and thus references to V at state I are not relevant to the calculation of these errors. Similarly, a define-undefine (DU) error occurs in the sequence from state I to state J if a variable V is defined at state I, undefined at state J, and referenced at no point in between.

An algorithm to locate DD and DU errors in an ordered set of state vectors is shown in Figure 16. In this algorithm, the set DEFSET holds the elements that have been defined but not yet referenced up to a given point of the sequence. As each state in the sequence is considered, the elements referenced in the statement following that state are removed from DEFSET, and, after checking for errors, the elements defined at that state are added to the set. A DD (DU) error occurs if the statement following the current state defines (undefines) an element that is in DEFSET.

When this algorithm locates either of these errors it announces the error, the variable that caused the error and the state at which the error was detected. This effectively substantiates the error, except for locating the state at which the errone-

```
Algorithm Locate_DD_And_DU_Errors(START, FINISH)
    (* START is the number of the first state to search
       FINISH is the number of the last state to search *)
  DEFSET := { e | e = Index($S^{START}$, v) for some v $\in$ Defs(Contents($S^{START}$, NS) ) }
    (* DEFSET is the set of variables that have been defined but not yet referenced *)
  For I := START+1 to FINISH
    N := Contents($S^I$, NS)  (* N is the number of the statement following state I *)
    DEFSET := DEFSET - {e | e = Index($S^I$, v) for some v $\in$ Refs(N) }
    If DEFSET $\cap$ {e | e = Index($S^I$, v) for some v $\in$ Defs(N) } $\neq$ {}
      then announce the variables corresponding to elements in the
                 intersection are used in a DD error at state I
    If DEFSET $\cap$ {e | e = Index($S^I$, v) for some v $\in$ UnDefs(N) } $\neq$ {}
      then announce the variables corresponding to elements in the
                 intersection are used in a DU error at state I
    DEFSET := DEFSET $\cup$ {e | e = Index($S^I$, v) for some v $\in$ Defs(N) }
  end (* for *)
end (* Algorithm Locate_DD_and_DU_Errors *)
```

**Figure 16. Algorithm Locate_DD_and_DU_Errors**

ous variable is first defined. This step, not shown in the algorithm, can be accomplished by either keeping each variable's defining point in DEFSET, or by a backward search of the execution history to find the definition point.

The final pair of Type 3 errors, dangling references and garbage are characterized as follows. As defined in subsection 4.3.1, an element that is associated with a variable of pointer type is a dangling reference if the address it contains is the address of no active element. Garbage cells are those active elements that are accessible from no pointer variable. To locate garbage or dangling references in a state, a garbage collection routine can mark all dynamically allocated elements that can be referenced through accessible pointers and can also verify that each pointer references some active element. When this has been done, any unmarked cell is garbage and a pointer that does not reference an active element is dangling. Substantiation of these errors requires announcing only the element that is in error. More sophisticated substantiation techniques are the subject of future research.

## 5.4 Relationship of Error Types and Execution Histories

Because execution histories and error types are both classified by their relation to computation termination, it is possible to define a relationship between execution histories and the error types. As shown in Figure 17, a Class A execution history is terminated by a Type 1 error; a Class B execution history has an infinite loop at it's end (i.e. a Type 2 error). Each of these errors is called the *base error* of its respective Class A or B execution history. Zero or more Type 3 errors can precede a base error in a Class A or Class B EH. Additionally, zero or more Type 3 errors can occur any-

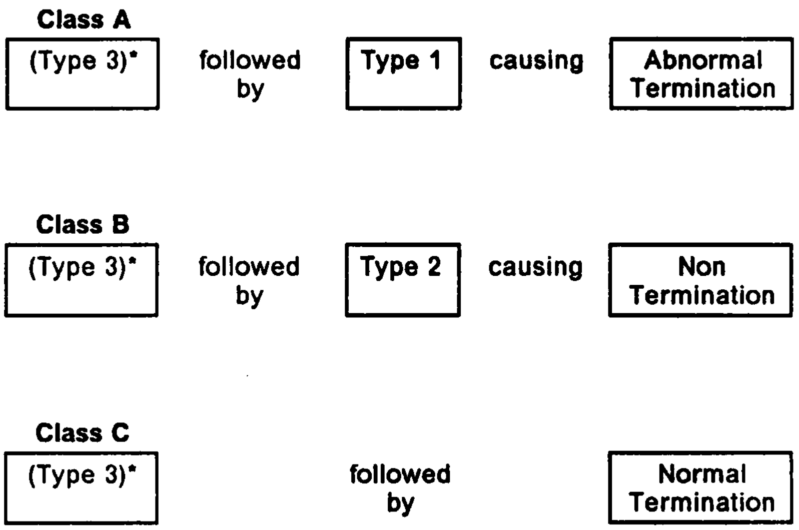


**Figure 17. Relation of Execution History Classes and Error Types**

where in a Class C execution history. This relationship of EH classes and error types provides a basis for an automatic debugging strategy. As developed in the next chapter, this strategy performs an execution history analysis that is guided by the EH class and by characteristics of detected errors.

# 6.0 One Approach to Automatic Debugging

The primary purpose of the automatic debugging system described in this dissertation is to analyze a program and its execution history and, based on this analysis, to provide the user with information on errors occurring in the program and on faults underlying these errors. Based on the EH classification and error analysis given in the previous chapters, a general approach for automatic debugging is to

- detect any Type 3 errors that may have caused a Type 1 or Type 2 error, and to
- apply specific fault localization techniques to appropriate errors and error combinations.

The most general implementation of this strategy would allow relationships among errors, error characteristics and fault localization techniques to be described by rules and facts in a knowledge base. Based on an existing program and an instance of its execution history characteristics, an inference engine could then apply these rules to automatically locate errors and apply fault localization techniques. This chapter describes a modest first step toward this goal of automatic debugging. The algorithm presented herein implements a particular automatic debugging strategy that is based on assumptions about the relative importance of errors and that uses a particular set of "hard-coded" fault localization heuristics. The next section describes this algorithm, and the following two sections respectively describe two aspects of the algorithm: slicing and the fault localization heuristics.

## 6.1 An Algorithm for Automatic Debugging

DEBUG, the algorithm for automatic debugging, is shown in Figure 18. A cursory examination of DEBUG's general logic shows that the algorithm's first task is to classify, in section (* 1 *), the execution history as Class A, B, or C. A Class A or a Class C execution history is easily detected by its abnormal or normal termination status. The infinite loop that characterizes a Class B execution history is detected by the State Match Algorithm given in Chapter 4. Based on the class of the EH, sections (* 2 *) and (* 3 *) of DEBUG search for errors in the execution history. As described in section 5.4 of Chapter 5, a Class A or Class B execution history contains zero or more Type 3 errors, followed respectively by a single Type 1 or Type 2 base error. Therefore, if the EH is either of Class A or Class B, then in section (* 2 *) DEBUG locates and substantiates the base error, and then in section (* 3 *) it searches for any Type 3 errors that may have an influence on the base error. To locate only those errors with influence, DEBUG confines its search to an EH slice that contains only states which impact the base error. As discussed in the next section of this chapter, the algorithm IndirectSlice calculates this slice, and also detects any incompatible pointers or corrupt elements whose use affects the base error. To locate

```
Algorithm DEBUG
(* This algorithm implements automatic debugging *)

(* Variables:
   CLASS - the class of the execution history
   IS - the indirect slice
   VSET - the set of variables in the slice criterion
   START - the number of the state used to start the slice  *)

Determine CLASS, the class of the execution history                          (* 1 *)
case CLASS of
A, B:  Substantiate the base error                                           (* 2 *)
       announce the errors detected by substantiation

       Determine VSET and START from the results of substantiation           (* 3 *)
       IS = IndirectSlice(VSET, START)
       announce any corrupt variables or incompatible pointers found in slicing
       for each state N in IS
          for each variable V in Refs(Stmt($S^N$, NS))
             if IsUndefined($S^N$, Index($S^N$, V)) or IsIncompatible($S^N$, Index($S^N$, V))
                then announce the error in variable V at state N

       if no Type 3 standards violations were found                          (* 4 *)
          then invoke Type_1_Heuristics or Type_2_Heuristics
             announce results of heuristic analysis

C:     for each state N in the execution history                             (* 5 *)
          for each variable V in Refs(Stmt($S^N$, NS))
             if IsUndefined($S^N$, Index($S^N$, V)) or IsIncompatible($S^N$, Index($S^N$, V))
                   or IsCorrupt($S^N$, Index($S^N$, V))
                then announce the error in variable V at state N

       if neither undefined nor incompatible errors were found               (* 6 *)
          then search for anomalies
             announce any anomalies that were detected
end  (* case *)
end  (* Algorithm DEBUG *)
```

**Figure 18. Algorithm DEBUG**

occurrences of other Type 3 errors, DEBUG explicitly searches for them in the states of the slice. If any Type 3 errors are found, they are assumed to underlie the base error, and consequently, DEBUG announces them to the debugging system and halts. If no Type 3 standards violations are found, then in section (* 4 *) DEBUG invokes one of two procedures (described in section 6.3 below) that use heuristics to localize code that is likely to be involved in the base error. The heuristics employed are not exhaustive; they are meant to only suggest possible analyses.

DEBUG uses a similar approach when working with a Class C execution history. Because there is no base error to guide a search, DEBUG searches the entire execution history for Type 3 errors; in section (* 5 *) it looks for standards violations, and if no violations are found, then in section (* 6 *) it looks for anomalies. Announcing any detected errors then completes the debugging process.

Based on the preceding description of DEBUG's general logic, the following discussion provides additional detail where needed. Section (* 1 *) needs no further explanation because the techniques used to classify the EH are fully described in Chapter 4. In the second section DEBUG uses the techniques described in the previous chapter to substantiate the base error of a Class A or Class B execution history. If the EH is of Class A, then the base error is of Type 1, and substantiation determines the precise expression that violated the standard and caused the error. For a Class B error the base error is of Type 2 and substantiation determines the exact states and statements involved in the infinite loop. Using this information, DEBUG then calculates, in section (* 3 *), an indirect slice of the execution history based on the location of the error, i.e. state START, and the variables involved, i.e. the variables in VSET. For a Type 1 error, START would be the terminal state of the EH and VSET would contain the variables that are in the expression that caused termination. For a Type 2 error, START would be the state that immediately precedes the first execution of the condition of the infinite loop (state START in Figure 15), and VSET would contain the variables used in the infinite loop's condition. The slice is calculated by algorithm IndirectSlice. As discussed in the next section of this chapter, this algorithm also locates any incompatible pointers or corrupt elements that can influence the base error. Other Type 3 errors are located by the explicit search of the slice, as shown in the latter part of section (* 3 *) of algorithm DEBUG. To conclude processing of Class A or Class B execution histories, section (* 4 *) of DEBUG applies fault localization techniques to the base error if no Type 3 errors were found. These techniques are the subject of the last section of this chapter.

As noted above, to process a Class C execution history, DEBUG first searches (in section 5) for Type 3 standards violations by examining each state in the EH and explicitly checking for the use of undefined, corrupt, or incompatible variables. Any errors that are found are announced to the debugging system. If none are found, DEBUG then uses the techniques described in the previous chapter to search the entire EH for Type 3 DD and DU anomalies and the final state of the EH for garbage or dangling references. Again, any anomalies detected are announced to the system.

This completes the discussion of algorithm DEBUG. The next section describes execution history slices and defines algorithms to compute them.

## 6.2 *Slices*

The concept of the execution history slice is based on the program slice developed by Weiser [WEIM82]. He defines a slice criterion to be a statement and a set of variables from a program, and a slice of that program on a given criterion to be the set of statements that can influence the computation of the given set of variables at the given statement. Extending this concept to state vector sequence execution histories, we define a slice criterion to be a state and a set of variables, and a slice of an execution history on a given criterion to be the set of states that influence the calculation of the given variables at the given state. A slice is a *direct slice* if the influence is *direct* and an *indirect slice* if the influence is direct or *indirect.* The state $S^M$ directly influences (i.e. has direct influence on) variable identifier V at state $S^N$ if either

1. M = State($S^N$, Index($S^N$, V) ), or
2. there exist a variable identifier W and a state U such that
    a. W $\in$ Refs(Contents($S^U$, NS)),
    b. M = State ($S^U$, Index($S^U$, W)), and
    c. state U+1 has direct influence on variable identifier V at state N.

Intuitively, point 1 states that M influences the criterion if V's value at state N is assigned at state M. Point 2 states that M influences the criterion if it influences a variable (W) that is used in a statement which precedes a state (U+1) that directly influences the criterion. In detail, point 2.a states that W is used in the statement that follows state U, 2.b that W's value in state U is most recently reflected in a previous state M, and point 2.c states that the state following state U directly influences the original criterion. As an example, consider the following program fragment.

```
(* Fragment One *)
                        state 101
1: X := 0
                        state 102
2: Y := X + 1
                        state 103
3: Z := X + 19
                        state 104
4: if Z < 0
                        state 105
5:    then A := Y + 1
6:    else A := Y + 2
                        state 106
```

In this code, states 102, 103 and 106 have a direct influence on variable A at state 106. Specifically,

**state 106** has direct influence on ({A}, 106) by point 1, that is, because 106 = State($S^{106}$, Index($S^{106}$, A));

**state 103** has direct influence on ({A}, 106) because for variable Y and state 105

- Y ∈ Refs(Contents($S^{105}$, NS)), i.e. Y ∈ Refs(6),
- 103 = State($S^{105}$, Index($S^{105}$, Y)), and
- state 106 has direct influence on ({A}, 106); and

**state 102** has direct influence on ({A}, 106) because for variable X and state 102

- X ∈ Refs(Contents($S^{102}$, NS)), i.e. X ∈ Refs(2),
- 102 = State($S^{102}$, Index($S^{102}$, X)), and
- state 103 has direct influence on ({A}, 106).

Thus the direct slice on the criterion ({A}, 106) is {102, 103, 106}.

In contrast, the indirect slice based on the same criterion is {102,103,104,105,106} because these states have direct or indirect influence on the criterion. Specifically, states 102, 103 and 106 have direct influence on the criterion, state 105 reflects the choice of the false branch of the **if**, and state 104 reflects the assignment of the variable Z which is used in the **if** statement. Thus, an indirect slice contains states that contribute to the values of variables in the criterion or that affect control flow which affect the calculation of those variables.

The formal definition of indirect influence requires first a definition of the function **MeToo**. This function takes as an argument the number of an arbitrary statement and returns a set of sequence numbers of conditional statements, i.e. either **if** or **while** statements, that contain statement n in their scope [LYLJ84]. In the example above, MeToo(6) = {4} because statement 6 is in the scope of statement 4. Similarly, MeToo(5) = {4}, but MeToo of statements 1, 2, 3, and 4 are all empty, because these four statements are in the scope of no conditional statement.

Based on this definition, state M has indirect influence on a variable identifier V at state N if

1. state M has direct influence on V at state N; or
2. there exists a state D such that
   a. Stmt($S^M$, NS) $\in$ MeToo(Stmt($S^D$, NS)), and
   b. state D has indirect influence on V at N; or
3. there exist a variable identifier W and a state U such that
   a. State M has indirect influence on W at state U,
   b. W $\in$ Refs(Contents($S^U$, NS), and
   c. state (U + 1) has indirect influence on V at N.

In the example above, states 102, 103 and 106 are included in the set of states that have indirect influence on variable A at state 106 because of point 1. Taking control flow into account, point 2 allows state 105 to be in the set because

- Stmt($S^{105}$, NS) $= 4 \in$ MeToo(Stmt($S^{106}$, NS)) $=$ MeToo(6) $= \{4\}$, and
- state 106 has indirect influence on ({A}, 106).

Finally, State 104 is included in the slice because of point 3. In detail it is included because for variable Z and state 104,

- state 104 has indirect influence on Z at state 104 (because 104 $=$ State($S^{104}$, Index($S^{104}$, Z)) and thus 104 directly influences Z at 104),
- Z $\in$ Refs(Contents($S^{104}$, NS)) $=$ Refs(4), and
- state 105 has indirect influence on A at state 106.

Based on these definitions we can define algorithms that calculate direct and indirect slices, but first we must consider how wild addressing operations, that is, the use of corrupt variables or incompatible pointers, affect a slice. Consider the motivation for slicing: to identify states that influence the base error of a Class A or Class B execution history. If a slice contains a state that is created by an assignment that uses a corrupt variable or incompatible pointer, then the states that influence the invalid value reflect the actions of statements that syntactically should have no influence. Thus the slice should not include these states. In the following example code, assume that pointer P is incompatible in statement 3 and that this assignment corrupts variable A.

```
(* Fragment Two *)
                        state 100
1: C := 19
                        state 101
2: A := 10
                        state 102
3: P↑.R := C+1             (* This assignment corrupts A *)
                        state 103
4: B := A + 1
                        state 104
5: D := B + 1
                        state 105
```

In this case, a naively taken slice on ({D}, 105) would include states 105, 104, 103 and 101 but not 102 which syntactically should be in the slice. Moreover, states 103 and 101, which influence the criterion through the wild address operation, should not be included. One method of excluding the influence of wild address operations is to calculate the slice and to then search for and exclude the influence of wild address operations. However, this could exclude a state that had influence through another variable, e.g. if statement 5 above were

5: D = B + C.

Therefore, the slicing algorithms explicitly test for corrupt variables and incompatible pointers at each state, and, if any are found, then any states that influence these variables at the given state are excluded from the slice.

Three algorithms for finding slices are shown in Figure 19 and Figure 20. The first algorithm, Slice, calculates a direct slice and ignores the effect of wild addressing. The second, DirectSlice, obviously also finds a direct slice, but the slice it calculates is modified by existing incompatible pointers and corrupt variables. The final algorithm, IndirectSlice, also considers wild addressing as it calculates an indirect

```
Algorithm Slice(VSET, N)
R = { }
For each variable identifier W in VSET
  I = State(S^N, Index(S^N, W))  (* I is the state where W's value is assigned *)
  R = R ∪ { I } ∪ Slice(Refs(Stmt(S^I, NS) ), I-1)
Return R
```

```
Algorithm DirectSlice(VSET, N)
R = { }
For each variable identifier W in VSET
  if IsCorrupt(S^N, Index(S^N, W) ) or
     IsIncompatible(S^N, Index(S^N, P) ) for some pointer expression P in W
  then  announce that W is corrupt or P is incompatible
  else I = State(S^N, Index(S^N, W))
       R = R ∪ { I } ∪ DirectSlice(Refs(Stmt(S^I, Index(S^I, W)) ), I-1)
return R
```

**Figure 19. Algorithms Slice and DirectSlice**

```
Algorithm IndirectSlice(VSET, N)
 R = { }
 For each variable expression W in VSET                                    (* 1 *)
      If IsCorrupt($S^N$, Index($S^N$, W) ) or
           IsIncompatible($S^N$, Index($S^N$, P) ) for some pointer expression P in W
      then  announce that W is corrupt or P is incompatible
      else I = State($S^N$, Index($S^N$, W))
            R = R ∪ { I } ∪ IndirectSlice(Refs(Stmt($S^I$, Index($S^I$, W)) ), I-1)     (* 2 *)
 For each M in MeToo( Stmt($S^N$, NS) )                                    (* 3 *)
      J := FindPrev(M, N)
      R := R ∪ { J } ∪ IndirectSlice( Refs( M ), J-1 )                     (* 4 *)
 Return R
end (* Algorithm IndirectSlice *)
```

**Figure 20. Algorithm IndirectSlice**

slice. Each of the algorithms calculates a slice based on the criterion (VSET, N) where VSET is a set of variables and N is the sequence number of a state.

The first algorithm, Slice, recursively adds to set R a slice that is based on the variable set, i.e. Refs(Stmt(S', NS)), and the state, i.e. I-1, that are used in defining each variable W in the criterion. This algorithm is sufficient if there is no wild addressing. In the presence of wild addressing, algorithm DirectSlice calculates a direct slice that excludes the effects of wild addressing by not making the recursive call for corrupt variables or incompatible pointers. In addition to calculating the direct slice, this algorithm also announces any corrupt variables or incompatible pointers that it finds.

The algorithm to find an indirect slice extends the direct slice algorithm by including all the states that indirectly influence the criterion. Algorithm IndirectSlice, shown in Figure 20, calculates an indirect slice based on criterion (VSET, N). In this algorithm, the function FindPrev(M, N) returns the number of the state that immediately follows the most recent execution of statement number M and that precedes state N. It can be calculated by a simple search of the EH backward from state N. Except for the recursive call to IndirectSlice in statement 2, the first loop (i.e. statement 1) of this algorithm is identical to DirectSlice. In the latter loop (i.e. statement 3) the algorithm considers each statement M in the MeToo set of the statement that precedes state N. For each such statement M, it first locates the state J that follows the most recent execution of statement M and then adds state J to the slice. It also adds an indirect slice based on the variables of the previous execution of statement M and on state J-1 which precedes that previous execution. Using program fragment one, defined earlier in this section, an indirect slice based on criterion ({A}, 106) would first add state 106 to the slice (by statement 2). The recursive call, IndirectSlice({Y}, 105), in statement 2 would then add states 103 and 102. In addition, statement 4 adds state 105 because statement 4 (of fragment one) is in MeToo($S^{106}$, NS) and 105 follows the most recent execution of statement 4 (of fragment one). Finally, the recursive call in statement 4, i.e. IndirectSlice({Z}, 104), adds states 102 and 104. This gives the desired slice, {102, 103, 104, 105, 106}.

As a final example of the indirect slice, in fragment two an indirect slice based criterion ({D}, 105) produces the set of states {104, 105}. States 104 and 105 are both added to the slice by statement 2 of IndirectSlice, 105 being added by the original call and 104 by a recursive call. Because variable A is corrupt and because the MeToo set of each statement in the fragment is empty, only the one recursive call is made and no additional states are added to the slice.

This concludes the discussion of direct and indirect slices. The next section concludes the chapter with a discussion of the heuristics for fault localization for Type 1 and Type 2 errors.

## 6.3 Fault Localization Heuristics

If the algorithm DEBUG does not locate any Type 3 errors in the indirect slice then it applies fault localization heuristics to the base error. The purpose of these heuristics is to use characteristics of the code and of the base error to locate conditions that may indicate the cause of the error. Where possible, specific code and variables are indicated as contributing to the error. In addition, certain pertinent results of EH analysis may be presented as a starting point for the user's own debugging analysis. The next two subsections respectively describe the heuristics that are applied to Type 1 and Type 2 errors.

### 6.3.1 Heuristics for Type 1 Errors

As shown in the algorithm in Figure 21, the heuristics that are applied to a Type 1 error depend on the base error's group (see statement 1). For a value error or a pointer error, there are two approaches to locating the variables and statements that may have influenced the error. The first approach (i.e. statements 2 and 3) uses DS, a direct slice based on the final state of the execution history and on the variables that are used in the statement preceding that state, and also uses VS, the set of variables that are assigned by statements that occur in DS. If the cardinalities of VS and DS are both one (see statement 2), then only one variable and one statement contributed to the error and the single state and single variable in these sets can be announced as having influenced the error. If the cardinality of VS is one but the cardinality of DS is greater than 1 (see statement 3), then a single variable has contributed to the error, but no statement that singlely influenced the error can be determined.

If more than one variable contributes to the error (i.e. $|VS| > 1$), then the second approach, shown in statement 4, can be applied: If the statement containing the error is in a loop and has executed successfully on a previous repetition of that loop, then the variables of the direct slice can be partitioned into two classes. The first class contains the variables whose most recent change precedes the previous (and successful) execution of the erroneous statement; the second contains the variables whose most recent change follows that statement. The variables in the second class are most likely to contribute to the error and the statements that change those variables may contain the fault underlying the error. To partition these variables, Type_1_Heuristics uses the function FindPrev(N, P) which, as described in Section

```
Algorithm Type_1_Heuristics

(* This algorithm uses heuristics to locate faults underlying a Type 1 error *)

(* Variables:

    VSET - the set of variables that are relevant to the error,
    STOP - the sequence number of the final state in the EH.
    GROUP - the group of the base error

    These variables are determined during the substantiation process *)

Determine VSET, STOP, and GROUP

N = Stmt($S^{STOP}$, NS)  (* N is the number of the statement that terminated.  *)
case GROUP of                                                            (* 1 *)
    value error, pointer error:
      DS = Direct_Slice(VSET, STOP)
      VS = {V|V $\in$ Defs(Stmt($S^M$, NS)) for some M in DS}
          (* VS is the set of variables defined by statements
                associated with states that are in DS          *)

      if |VS| = 1  and  |DS| = 1                                         (* 2 *)
          then (* only one variable and one assignment influence the error *)
              announce the variable and statement that influence the error
      else if |VS| = 1  and  |DS| > 1                                    (* 3 *)
          then (* only one variable and several assignments influence the error *)
              announce the variable and statements that influence the error
      else if |VS| > 1  and
              $S^{STOP}$ occurs in a loop in which statement N executes earlier,   (* 4 *)
                i.e. if FindPrev(N, STOP) $\neq$ UNDEFINED
          then for each V in VS
              if State($S^{STOP}$, Index($S^{STOP}$, V)) < FindPrev(N, STOP)
                  then announce V changes before the
                              previous execution of statement N
                  else announce V changes at or after the
                              previous execution of statement N
```

**Figure 21. Algorithm Type_1_Heuristics: (Continued on Next Page)**

```
Algorithm Type_1_Heuristics (Continued)

availability error:

    if statement N is a call to New then
        For each V that corresponds to an identified variable in state STOP,        (* 5 *)
            calculate the frequency distribution of the calls to New that
            allocate the identified variables using FindAlloc($S^{STOP}$, Index($S^{STOP}$, V)).

        announce for each New statement that creates an element                     (* 6 *)
            that is active in state STOP: (1) the number of elements it
            allocates per record, and (2) the number of records it has created that
            are still active in state STOP.

        announce any New statements that either allocates a record                  (* 7 *)
            greater than MAXNSIZE elements or had more than MAXNCALLS calls
            that created currently active elements.

    else (* error occurred during a call of a procedure other than New *)

        Using the return addresses found on element NS of state STOP,               (* 8 *)
            calculate the frequency distribution of the procedure invocation
            statements that are still executing.

        announce for each statement that calls a procedure which                    (* 9 *)
            has not terminated at state STOP: (1) the number of cells
            allocated during a call of that procedure and (2) the number of
            unterminated instances of that call statement.

        announce any unterminated Call statement that either                        (* 10 *)
            allocates more than MAXPSIZE cells or has more than MAXPCALL
            unterminated instances.
end (* case *)
end (* Algorithm Type_1_Heuristics *)
```

**Figure 21. Algorithm Type_1_Heuristics: (Continued)**

6.2, locates the state that precedes state P and that immediately follows the previous execution of statement N, if such a previous execution exists. Using this function, each variable V in VS is tested in the comparison

$$\mathrm{State}(S^{STOP}, \mathrm{Index}(S^{STOP}, V)) < \mathrm{FindPrev}(N, STOP)$$

which compares the defining point of variable V with the point of the previous execution of statement N. If this comparison is true, then the algorithm announces that V was changed before the previous execution of statement N, and if false it announces that V was changed after the previous execution. After each variable has been tested, the algorithm terminates.

If the base error is an availability error, then it is assumed to occur either during the execution of a **new** statement or during the call of some user defined procedure. Statements 5, 6, and 7 of Type_1_Heuristics deal with errors that occur during execution of a **new** statement. Two possible causes of this error are considered: either the statement is executed too many times because it is in a loop that does not terminate properly, or the record being allocated is too large. To test for these conditions, statement 5 of the algorithm calculates the frequency distribution of the calls to each **new** statement. In statement 6 the size and frequency of each call are announced, and in statement 7, any call that exceeds some predefined limit in frequency (MAXNCALLS) or size (MAXNSIZE) is also announced. Establishment of these limits is left as an implementation consideration. The final statements of this algorithm, 8, 9, and 10, handle an availability error that occurs during a call of a user defined procedure. This error is handled similarly to the previous one, except that the frequency distribution (see statement 8) is of procedure calls that have current activations, i.e. that are still executing, rather than of **new** statements. As above, the distribution and size of each call is announced (see statement 9) along with any call that exceeds the limits MAXPSIZE and MAXPCALL (see statement 10).

This completes the definition of the heuristic techniques for Type 1 errors. The following section describes the techniques for errors of Type 2.

### 6.3.2 Heuristics for Type 2 Errors

As in the case of Type 1 errors, there are two heuristic approaches for locating code and variables that may contribute to a Type 2 error. As shown in statements 1 through 7 of Figure 22 the first approach is to determine if the infinite loop was caused by a loop control variable that does not change. To do this, Algorithm Type_2_Heuristics must calculate the values of FIRST and LAST, the sequence numbers of the first and last statement in the infinite loop, and the values of START and FINISH, the numbers of the first and last states in the first cycle of the infinite loop. The substantiation algorithm given in subsection 5.3.2 fully describes these variables and details their calculation. Also calculated by Type_2_Heuristics are the four sets described below.

- LCV is the set of variable that are used in the loop's condition. As seen in statement 1, Refs(FIRST) calculates this set of variables.
- DFV is the set of variables that can be defined by some statement which appears within the loop but which may not be executed. DFV is calculated in statement 2 as the union of the sets of variables that are in Defs of each statement that appears in the infinite loop.
- MDV is the set of variables that are modified but that do not necessarily change as they are assigned by some statement which is executed within a cycle of the

```
Algorithm Type_2_Heuristics

(* This algorithm uses heuristics to locate possible faults underlying
   a Type 2 base error *)

(* Variables whose values are determined during error substantiation:

    FIRST - the number of the first statement in the infinite loop
    LAST - the number of the last statement in the infinite loop
    START - the state preceding statement FIRST's first execution
            in the infinite part of the EH
    STOP - the last state of the cycle that begins with state START

PREVSTART - state preceding statement FIRST and used for comparison.
            Found using algorithm FindPREVSTART.

Sets:
    LCV - variables in the infinite loop's condition
    DFV - variables that are defined in the infinite loop
    MDV - variables that are modified in the loop cycle
    CHV - variables whose values change in the loop cycle   *)

LCV = Refs(FIRST)                                                       (* 1 *)
DFV = $\bigcup$ Defs(I), for FIRST $\leq$ I $\leq$ LAST                 (* 2 *)
MDV = {V|START < State($S^{STOP}$, Index($S^{STOP}$, V)) < STOP }       (* 3 *)
CHV = {V | V $\in$ MDV and there is an integer I such that START < I < STOP    (* 4 *)
        and Contents($S^I$,Index($S^I$, V) $\neq$ Contents($S^{I+1}$, Index($S^{I+1}$, V)
if LCV $\cap$ DFV = {}                                                  (* 5 *)
    then announce No loop control variable appears on the left
             side of an assignment within the loop

else if LCV $\cap$ MDV = {}                                             (* 6 *)
    then announce No loop control variable is modified within the loop

else if LCV $\cap$ CHV = {}                                             (* 7 *)
    then announce No loop control variable changes within the loop

else if the statements of the infinite loop are nested within another loop     (* 8 *)
    then if FindPREVSTART finds an earlier, finite repetition of the loop
        then announce the variables that differ between states START and PREVSTART
end (* Algorithm Type_2_Heuristics *)
```

**Figure 22. Algorithm Type_2_Heuristics**

loop. As seen in statement 3, any variable whose value is assigned at a state between states START and STOP is in this set.

- CHV is the set of variables that are actually changed within the cycle. A variable is in this set if it was modified in the cycle and if there is a state (state I in statement 4) such that V's values at states I and I+1 are different.

Our initial heuristic is based on the assumption that the intersection of LCV with each of the latter 3 sets should yield all non-empty sets. If it does not, then the loop control variable can not be or is not changed within the loop, which is a fault. Accordingly, if the intersection of LCV and DFV is empty then the algorithm announces that none of the loop control variables appear on the left side of an assignment within the loop (see statement 5). If that intersection is nonempty, but the intersection of LCV and MDV is empty (see statement 6) then no statement that assigns a value to a loop control variable is executed within the loop, and the algorithm announces that no loop control variable is modified within the loop. Finally, if neither of the previous two cases hold, but the intersection of LCV and CHV is empty (see statement 7), then none of the loop control variables changes over the cycle. In this case it is announced that no loop control variable changes within the loop.

If the intersection of LCV with each of the three sets is non-empty, then, the second approach to localizing the fault underlying the infinite loop is used. This approach applies if the statements of the non-terminating loop (statements FIRST through LAST in Figure 23) are nested within a larger loop (statements OUTER to EOUTER), and if the inner loop executed and terminated normally in a previous pass (i.e. in the pass between states OREP1 and OREP2 in Figure 23) of the outer loop. If both conditions apply, then the variables that are most likely to be relevant to the error are those whose values differ between states PREVSTART (which reflects the

**Illustrative Source Code**

```
OUTER       (* First statement of the nesting loop *)
 . . .
    FIRST   (* First statement of the nested, infinite loop *)
 . . .
    LAST    (* Last statement of the nested, infinite loop *)
 . . .
EOUTER      (* Last statement of the nesting loop *)
```

**Illustrative Execution History**

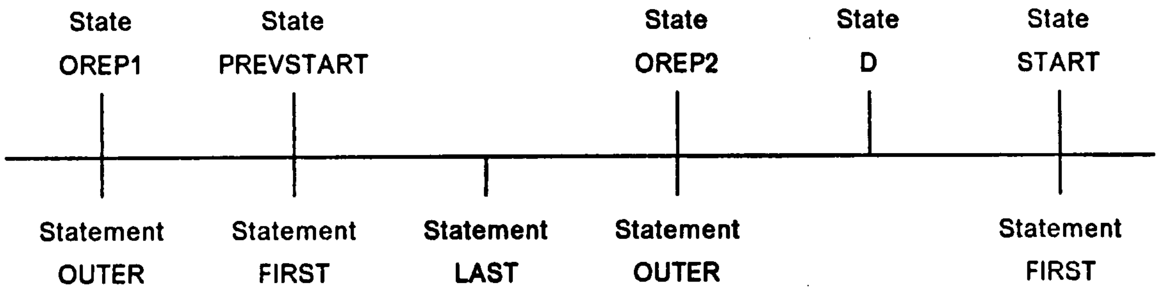


**Figure 23. Illustrative Code and Execution History for Type 2 Heuristics**

first execution of statement FIRST during the finite repetition) and START (the initial state of the infinite cycle).

Statement 8 of Type_2_Heuristics implements this approach. If the non-terminating loop is nested (i.e. if the condition in the first IF of statement 8 of Type_2_Heuristics is true), then, in the second IF, algorithm Find_PREVSTART attempts to locate state PREVSTART. If the state exists, then states START and PREVSTART are compared and variables that differ between them are announced as relevant to the error.

The code for algorithm FindPREVSTART is as follows.

```
Algorithm FindPREVSTART
    OREP2 := FindPrev(OUTER, D)                                   (* 1 *)
    If OREP2 <> UNDEFINED
       then OREP1 := FindPrev(OUTER, OREP2)                       (* 2 *)
            If OREP1 <> UNDEFINED
               then PREVSTART := FindNext(FIRST, OREP1)           (* 3 *)
```

Statement 1 of this algorithm searches backward from state D (the first state of the infinite portion of the EH) to find state OREP2 which reflects the last execution of statement OUTER before the infinite part of the computation. If this state exists (i.e. if OREP2<>UNDEFINED) then statement 2 locates state OREP1 which reflects the previous execution of statement OUTER. If this state is found, then in statement 3 the function FindNext locates state PREVSTART which reflects the first execution of statement FIRST after state OREP1. FindNext(M ,N) operates analogously to FindPrev(M, N) except that it locates the state that follows the *next* execution of statement M *after* state N rather than the *previous* execution of statement M *before* state N.

This concludes the discussion of the heuristic analysis of the EH and of the scheme for automatic debugging. The following chapter summarizes the main accomplishments of the dissertation and discusses future work.

# 7.0 Summary and Future Work

Chapters 2 through 6 of this dissertation describe a theoretical basis for the development of an automatic execution history based debugger. To conclude the dissertation, section 7.1 of this chapter summarizes the main results of the research, and section 7.2 outlines areas remaining for future work.

## 7.1 Summary

The main results of this disseration are three in number:

- a state vector model for execution history representation and analysis,
- an analysis of errors as characterized within the state vector model, and
- a strategy for automatic debugging, based on error occurrence.

Each result is summarized below.

As discussed in Chapter 2, the state vector model represents the execution of programs that are written in procedure oriented, imperative languages. These languages are organized into a hierarchy: Level 1 languages have elementary data types and the simple control structures of sequence, if-then-else and while loops. Level 2 languages incorporate procedures, and those of Level 3 add dynamic variables. Based on this language hierarchy, Chapter 3 defines the state vector characteristics required for modeling programs written in each language level. Chapter 4 defines the state transition functions that model execution of individual statements. Based on these definitions, Chapter 4 then defines the state vector sequences that model program execution and that serve as an execution history. As seen in Chapter 3, Level 1 languages require fixed length vectors whose elements are 3-tuples consisting of a content value and state and statement indicators. Chapters 5 and 6 specify how this information aids error characterization and the automatic debugging process. Level 2 languages require fixed length vectors whose elements can be stacks of 3-tuples, while Level 3 languages require a vector of infinite length whose elements are stacks of 4-tuples. Each 4-tuple augments the 3-tuple with the address of the storage location that is allocated to the variable associated with the 4-tuple. This addition allows the modeling of wild address errors where an element is illegally accessed by a pointer that has an errant value.

In addition to defining three levels of state vector characteristics, Chapter 4 defines functions, called action functions, that modify the state vector by either (a) replacing a single n-tuple within an element of a state vector, (b) pushing or popping a single cell (i.e. n-tuple) onto or off of an element, or (c) changing the number of active (i.e. currently in use) elements of a state vector. These functions as well as characteristics of the state vector are summarized in Figure 5.

To complete the state vector model, Chapter 4 uses the action functions to define state transition functions that model the state to state transitions which represent the execution of individual program statements. An assignment statement is modeled using the action functions that replace an individual state vector element. Similarly, the Level 1 control structures are modeled by assigning the sequence number of the next statement to be executed to element NS, the element that indicates the statement which is next in the execution sequence. Procedure call and return are modeled by pushing and popping the procedure's parameters, local variables, return address and entry point onto and off of the appropriate cells. Finally, dynamic variable allocation and deallocation are modeled by the functions that respectively increase and decrease the number of active elements in the state vector. Since each state vector to state vector transition models execution of a single statement, a sequence of statements then models execution of part or all of a program, and thus a state vector sequence serves as an execution history. Chapter 4 defines these state vector sequence execution histories and concludes with a EH classification scheme that differentiates state vector sequences by their termination status. In this scheme, execution histories that terminate abnormally are of Class A, those that do not terminate are of Class B, and Class C histories terminate normally.

Based on this EH model, the analysis of errors discussed in Chapter 5 (a) classifies errors based on their termination status, (b) outlines the characterization, detection and substantiation of errors, and (c) describes the relation between the class of an EH and the errors that appear within it. Considering first the classification scheme, errors that cause immediate termination are classed as Type 1. These errors include value, pointer and availability errors. Type 2 errors are those that prevent termination, i.e. infinite loops, while Type 3 errors are those that do not affect termination, i.e. anomalies and certain standards violations. Following the grouping provided by this classification scheme, Chapter 5 outlines the execution history characteristics that distinguish each error and describes techniques for detecting such characteristics, and thus the errors, within an execution history. This chapter also discusses techniques for substantiating an error, i.e. determining details of a particular instance of that error. In conclusion it indicates the relationship between the EH classes and the error types. This relationship, i.e. the existence of a single Type 1 or Type 2 base error in a Class A or Class B EH, respectively, and the existence of zero or more Type 3 errors in any class EH, provides the structure for the automatic debugging algorithms defined in Chapter 6.

Using this EH class and error type relationship, algorithm DEBUG, described in Chapter 6, implements a particular strategy for automatic debugging. In this strategy, DEBUGS's initial choices are based on the class of the EH, and later actions are guided by assumptions about the relative importance of errors. In processing a Class A or Class B EH (i.e. one that does not terminate normally), DEBUG first substantiates the base error and then searches for Type 3 errors that occur in an indirect EH slice relative to the base error. (Algorithms to calculate slices are discussed in Chapter 6.) If such an error is found, then DEBUG assumes that this error must be corrected before further debugging can occur, and it does no more analysis. If no Type 3 errors are found in the slice, then DEBUG searches for specific source code and execution history characteristics that can indicate the cause of the base error. These characteristics are implicitly defined in Algorithms Type_1_Heuristics and Type_2_Heuristics. If no such characteristics are found, then DEBUG's has exhausted its options in evaluating Class A and B execution histories and it terminates. Processing of a Class C EH is similar to that just discussed, except that because an EH of this type has no base error to serve as a basis for a slice, DEBUG performs its searches over the entire EH. Assuming that a Type 3 standards violation is more important than a Type 3 anomaly, DEBUG first searches for violations, and if none are found then it performs a search for anomalies. In either case, when the search is complete, the algorithm terminates.

Although DEBUG's capabilities are limited, it does provide a starting point for research in execution history based debugging. Obviously more powerful and sophisticated techniques are needed. Aspects of possible improvements are the subject of future work and are discussed in the concluding section of this chapter.

## *7.2 Future Work*

The research results outlined in this dissertation provide the theoretical and algorithmic foundation for developing an intelligent debugger. Leading to the construction of this debugger is further research in the following areas:

1. developing a prototype debugger that can be used to test techniques for EH representation and collection,
2. applying the execution history model to the representation of features of an existing programming language,
3. expanding the set of error detection and fault localization techniques,
4. representing source code and EH information and the error detection and fault localization techniques within a suitable knowledge base, and developing an inference engine to use with that knowledge base,
5. designing smart commands, and
6. developing a graphics-based, multi-view user interface.

Each of these areas is discussed in one of the subsections below.

### 7.2.1 Developing a Prototype Debugger

In developing a prototype debugger, research will concentrate on solving practical problems associated with collecting, representing and analyzing the execution history as well as implementing the automatic debugging algorithms outlined in Chapter 6. All information about static and dynamic program characteristics will be hard coded into the program as will be the automatic debugging techniques; no general inference capabilities will be provided. Construction and subsequent use of this prototype will provide valuable insight in three areas. First, it will verify the error detection algorithms, second, it will provide necessary experience in execution history collection and analysis, and third, it will demonstrate the efficacy of the approach.

The primary problem in constructing this prototype, collecting the execution history, has the following possible solutions.

1. Modified compilers can support collection of a program's execution history by inserting appropriate instructions at strategic points in the object code of the suspect program. When executed, these instructions record the program's run time behavior.
2. Similar to the modified compiler approach, preprocessors can be used to instrument the source code of the suspect program. Again, these additional instructions record the program's runtime behavior.
3. Finally, the debugger itself can provide its own interpretive system in which the data collection capability is built into the interpreting procedures.

Clearly, each of the above approaches has advantages and disadvantages. For building a prototype, approach (2) appears to provide the necessary information most simply.

In addition to choosing an execution history collection method, a second question must be resolved: What pieces of data are actually recorded. That is, is every state vector stored in its entirety? Approaches discussed in the literature favor saving only changed values [BALR69, FAIR75]. Additional investigation should indicate the applicability of this approach to our model of computation and system structure.

### 7.2.2 Applying the Model to an Existing Language

A second area of future research involves representing the features of an existing programming language within the state vector sequence model of execution history. Because it is well known, has non-trivial features and is not overly complex, Pascal has served as a basis for discussion in this dissertation. For the same reasons it is a logical choice for this research area.

Chapter 4 lists several language features whose modeling is not covered in this dissertation. Most of these features can be represented with no change in the state vectors and with no additional action functions. The **repeat** loop, for example, can be modeled by a state transition function that models the branch from the test to either the first statement of the loop or the loop statement's successor, depending on the value of the loop's test. This function would effectively be identical to the function that models the **while** loop, but the true and false destinations of the loop test would of course be reversed. In contrast to the ease of representing **repeat** loops, modeling some language features may require modification of the underlying model. Input and output, for example, could be modeled by including in each state a pair of files that represent the current condition of the input and output processing. Such a modification would not greatly affect the modeling of the language features defined in this dissertation, but it could have considerable effects on the assumptions underlying debugging techniques. These effects must be explored and additional techniques developed to exploit the new model characteristics.

### 7.2.3 Expanding the Set of Debugging Techniques

The error detection and fault localization techniques described in Chapters 5 and 6 illustrate an approach to automatic debugging. Because these techniques are not meant to be exhaustive, developing a production quality debugger will require additional research in this area.

One unexplored area involves debugging pointer errors. The existing techniques outlined in Chapters 5 and 6 detect garbage and dangling references, but they make no attempt to locate the statements at which these conditions are created. The debugging techniques should be expanded to include this search. Similarly, the slicing algorithms discussed in Chapter 6 include code that locates wild address operations (i.e. the creation or use of corrupt or incompatible elements), but no additional analysis of the conditions is given. Any wild address error is directly or indirectly caused by an allocation that illegally and inadvertently makes a dangling reference incorrectly point to an object. Detection of this allocation is essential to debugging pointer errors.

Existing automatic debugging methods for other error groups can similarly be extended. New techniques may incorporate the knowledge of the source code structure in which the error occurs and may consider combinations of errors in addition to single errors. Incorporating these additional techniques will be facilitated by the modifications described in the next subsection.

### 7.2.4 Incorporating a Knowledge Base and an Inference Engine

The algorithms of Chapters 5 and 6 are suitable for illustrating and testing concepts in automatic debugging, but because they are hard coded they are limited in scope and difficult to modify and extend. An approach that allows easier modification and expansion is to express the debugging algorithms within a knowledge base and to use an inference engine to drive the algorithms and their application. As basic information, the knowledge base should contain the suspect program's source code, an instance of the program's execution history, and additional facts (i.e. the EH's termination status) about both. In addition, it should contain rules that express the actions required by the error detection and fault localization algorithms, and the conditions under which these rules are to be applied and the actions performed. Given a configuration of the knowledge base, an inference engine can apply appropriate rules and can derive additional information about errors that exist within the EH and about faults that underlie these errors. Given, for example, an abnormally terminated EH, one rule could specify that the base error should be substantiated and a slice be calculated. Additional rules could specify the actions to be taken based on information gained from taking the slice.

To create this knowledge base, research is required to determine the information that it must contain and how this information is to be represented. In conjunction with these decisions is the development of an inference engine that understands the facts, rules, and actions expressed within the knowledge base so that it can (a) determine what rules apply in a given configuration of the knowledge base and (b) perform the actions specified by those rules. Completion of these tasks will allow the expression of more varied and powerful techniques and will enable convenient expansion and modification of the debugging system.

### 7.2.5 Designing Smart Commands

As outlined in Chapter 1, our debugging system is meant to perform an initial, automatic execution history analysis and report its findings to the user. The results may indicate the detection of errors, faults underlying the errors, or perhaps nothing. In the event that the results of the automatic analysis are not satisfactory, then the user can assume control of the debugging process and use "smart commands" to perform a manual analysis. Example smart commands include

- calculating either a direct or an indirect EH slice based on a criterion of a given state and set of variables,
- calculating a program slice [WEIM82],
- displaying the range of values taken by a variable over a range of statements,
- showing a "flowback" analysis [BALR69] of a given variable at a given statement,
- displaying frequency counts of specified statements and statement groups,
- indicating flow of control through a given section of code, and
- graphically displaying dynamically allocated structures.

Other commands can be developed in further research. Implementing these commands involves (a) designing specific algorithms for each, and (b) representing each within the knowledge base described above. Invoking the commands will be facilitated through the user interface described in the next section.

### 7.2.6 Developing a User Interface

In execution history based debugging the amount of information available can overwhelm the user. Thus a final aspect of future research is the development of a graphics-based, multi-view interface that can effectively display the complex information available from the EH automatic analysis and from the smart commands. This interface should support several visual abstractions of both the source code and the execution history. Example program abstractions include pages and slices of the program text and graphic representations of the program structure. Each of these essentially static views can be supplemented with dynamic information available from the EH, e.g. a program slice can include only statements that have actually been executed. In addition to supplementing the static program views, the EH itself can be abstracted and displayed; example EH abstractions include direct and indirect slices and displays of dynamically allocated structures. Existing program development and debugging environments such as Omni [ARTJ84], Pecan [REIS84], Pict [GLIE84] and VIPS [ISOS87] provide static views. Augmenting the methods developed for these systems with techniques for presenting static views will support the multiple views needed for EH based debugging.

## 7.3 Conclusion

The execution history model, error analysis, and automatic debugging techniques described in this dissertation provide a foundation for developing an intelligent EH based debugger. Further research will substantiate these results and will provide the practical techniques and theoretical results necessary to design and construct a complete intelligent debugging system based on this foundation.

# References and Bibliography

[ACM83] ACM Proceedings ACM Sigsoft/Sigplan Software Engineering Symposium on High-Level Language Debugging, *SIGPLAN Notices*, Vol. 18, No. 8, August 1983, 208 pp.

[ALBC84] Alberga, C. N., Brown, A. L., Leeman, G. B., Mikelsons, M., and Wegman, M. N., Program Development Tools, *IBM Journal of Research and Development*, Vol. 28, No. 1, January 1984, pp. 60-73.

[ALLF74] Allen, F. E., Interprocedural Data Flow Analysis, *Proc. IFIP Congress 1974*, North Holland Publ. Co, Amsterdam, The Netherlands, 1974, pp. 398-402.

[ALLF76] Allen, F. E. and Cocke, J., A Program Data Flow Analysis Procedure, *Communications of the ACM*, Vol. 19, No. 3, March 1976, pp. 137-147.

[ARTJ84] Arthur, J.D. and Comer, D.E., Omni: An Interactive Programming Environment Based on Tool Composition, *Proceedings of the IEEE Computer and Applications Conference*, Chicago, IL, November 1984, pp. 28-36.

[AT&T84] AT&T, A Symbolic Debugging Program - SDB, *UNIX System V Programming Guide*, April 1984, pp. 7.1-7.16.

[BALR69] Balzer, R.M., EXDAMS – EXtendable Debugging and Monitoring System, *AFIPS Spring Joint Computer Conference*, Vol. 34, 1969, pp. 567-580.

[BARD63] Barron, D.W. and Hartley, D.F., Techniques for Program Error Diagnosis on EDSAC-2, *The Computer Journal*, Vol. 6, No. 1, April 1963, pp. 44-49.

[BASV84] Basili, V. R. and Perricone, B. T., Software Errors and Complexity: an Empirical Investigation, *Communications of the ACM*, Vol. 27, No. 1, January 1984, pp. 42-52.

[BEAB83] Beander, B., VAX DEBUG: An Interactive, Symbolic Multilingual Debugger, *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on High-Level Debugging*, March 1983, pp. 173-179.

[BOEB73] Boehm, B., Software and Its Impact: A Quantitative Assessment, *Datamation*, Vol. 19, No. 5, May 1973, pp. 48-59.

[BOHC66] Bohm, C. and Jacopini, G., Flow Diagrams, Turing Machines, And Languages With Only Two Formation Rules, *Communications of the ACM*, Vol. 9, No. 5, May 1966, pp. 366-371.

[BONA69] Bond, A. H, Rightnout, J., and Steven, L., An Interactive Graphical Display Monitor in a Batch-Processing Environment with Remote Entry, *Communications of the ACM*, Vol. 12, No. 11, November 1969, pp. 595-603, 607.

[BROJ78] Browne, J.C. and Johnson, D.B., FAST: A Second Generation Program Analysis System, *Proceedings of the Third International Conference on Software Engineering*, May 1978, pp. 142-148.

[CART84] Cargill, T. A., Debugging C programs with Blit, *AT&T Bell Labs Technical Journal*, Vol. 63, 8 (part 2), October 1984, pp. 1633-1647.

[CART85] Cargill, T. A., Implementation of the Blit Debugger, *Software - Practice and Experience*, Vol. 15, No. 2, February 1985, pp. 153-68.

[CHAB81] Chandrasekaran, B. and Radicchi, S. (Ed.), *Computer Program Testing*, 6 July 1981.

[CHAC83] Charlton, C. C. and Leneg, P. H., Aids for Pragmatic Error Detection, *Software - Practice and Experience*, Vol. 13, No. 1, January 1983, pp. 59-66.

[CHUA41] Church, A., The Calculi of Lambda-Conversion, *Annals of Mathematical Studies*, No. 6, 1941.

[CLAL85] Clark, L.A. and Richardson, D.J., Applications of Symbolic Evaluation, *The Journal of Systems and Software*, Vol. 5, No. 17, February 1985, pp. 15-35.

[CRAS85] Crawford, S. G., McIntosh, A. A., and Pregibon, D., An Analysis of Static Metrics and Faults in C Software, *The Journal of Systems and Software*, Vol. 5, No. 1, February 1985, pp. 37-48.

[CUFR72] Cuff, R.N., A Conversational Compiler for Full PL/1, *Computer Journal*, Vol. 15, No. 2, May 1972, pp. 99-104.

[CURR84] Curtis, R. and Wittie, L., Global Naming in Distributed Systems, *IEEE Software*, Vol. 1, No. 3, 1984, pp. 76-80.

[DONJ76] Donahue, J. E., *Complimentary Definitions of Programming Language Semantics*, Lecture Notes in Computer Science, Vol. 42, Springer-Verlag, 1976, 172 pp.

[DUNT64] Dunn, T.M. and Morrissey, J.H., Remote Computing - An Experimental System, *Proceedings of the AFIPS Spring Joint Computer Conference*, Vol. 25, 1964, pp. 413-424.

[ELGC64] Elgot, C. C. and Robinson, A., Random-Access Stored-Program Machines, an Approach to Programming Languages, *Journal of the ACM*, Vol. 11, No. 4, October 1964, pp. 365-399.

[ELLB82] Elliott, B., A High-level Debugger for PL/I, Fortran and Basic, *Software - Practice and Experience*, Vol. 12, No. 4, April 1982, pp. 331-340.

[ENDA75] Endres, A., An Analysis of Errors and Their Causes in System Programs, *IEEE Transactions on Software Engineering*, Vol. SE-1, No. 2, June 1975, pp. 140-149.

[ENGG83] Engels, G., Pletat, U., and Ehrich, H.-D., An Operational Semantics for Specifications of Abstract Data Types with Error Handling, *Acta Informatica*, Vol. 19, Fasc. 3, July 1983, pp. 235-253.

[EVAT66] Evans, T. G. and Darley, D. L., On-Line Debugging Techniques: A Survey, *AFIPS Fall Joint Computer Conference*, Vol. 29, Fall 1966, pp. 37-50.

[FABR65] Fabry, R.S., MADBUG - A MAD Debugging System, *The Compatible Time Sharing System, A Programming Guide*, Second Edition, MIT Press, Cambridge, Mass., 1965.

[FAIR75] Fairley, R., An Experimental Program Testing Facility, *IEEE Transactions on Software Engineering*, Vol. SE-1, No. 4, December 1975, pp. 350-357.

[FAIR81] Fairley, R. E., Software Testing Tools, *Computer Program Testing*, Chandrasekaran and Radicchi (Eds.), 1981, pp. 151-186.

[FLOR67] Floyd, R. W., Assigning Meaning to Programs, *Proceedings of Symposia in Applied Mathematics*, Vol. 19, 1967, pp. 19-32.

[FOSL76] Fosdick, L.D. and Osterweil, L.J., Data Flow Analysis in Software Reliability, *ACM Computing Surveys,* Vol. 8, No. 3, September 1976, pp. 305-330.

[FOUJ84] Fouet, J., An Expert System for the Manipulation of Programs, *Proceedings of the 4th Jerusalem Conference on Information Technology,* May 1984, pp. 460-467.

[GAIJ85] Gait, J., A Debugger for Concurrent Programs, *Software - Practice and Experience,* Vol. 15, No. 6, June 1985, pp. 539-554.

[GAUE82] Gauss, E. J., The "Wolf Fence" Algorithm for Debugging, *Communications of the ACM,* Vol. 25, No. 11, November 1982, p. 780.

[GETS83] Getz, S. L., Kalligiannis, G., and Schach, S. R., A Very High: Level Graphical Trace for the Pascal Heap, *IEEE Transactions on Software Engineering,* Vol. SE-9, No. 2, March 1983, pp. 179-185.

[GIER83] Giegerich, R., A Formal Framework for the Derivation of Machine-Specific Optimizers, *ACM Transactions on Programming Languages and Systems,* Vol. 5, No. 3, July 1983, pp. 478-498.

[GILD84] Gilmore, D. J. and Smith, H. T., An Investigation of the Utility of Flowcharts During Computer Program Debugging, *International Journal of Man-Machine Studies,* Vol. 20, No. 4, April 1984, pp. 357-372.

[GLAR80] Glass, R. L., Real Time: The "Lost World" of Software Debugging and Testing, *Communications of the ACM,* Vol. 23, No. 5, May 1980, pp. 264-271.

[GLAR81] Glass, R. L., Persistent Software Errors, *IEEE Transactions on Software Engineering,* Vol. SE-7, No. 2, March, 1981, pp. 162-168.

[GLAR82] Glass, R. L., Real-time Checkout: The 'Source Error First' Approach, *Software - Practice and Experience,* Vol. 12, No. 1, January 1982, pp. 77-83.

[GLIE84] Glinert, E.P., and Tanimoto, S.L., Pict: An Interactive Graphical Programming Environment, *IEEE Computer,* Vol. 17, No. 11, November 1984, pp. 7-25.

[GOEA80] Goel, A. L., Software Error Detection Model with Applications, *The Journal of Systems and Software,* Vol. 1, No. 3, 1980, pp. 243-249.

[GOOJ75] Goodenough, J.B. and Gerhart, S.L., Towards a Theory of Test Data Selection, *IEEE Transaction on Software Engineering,* Vol. SE-1, No. 6, June 1975, pp. 156-173.

[GOOJ77] Goodenough, J.B., and Gerhart, S.L., Toward the Theory of Testing: Data Selection Criteria, *Current Trends In Programming Methodologies*, Vol.2, R.T. Yeh (Ed.), Prentice-Hall, 1977, pp. 44-79.

[GREI59] Greenwald, I.D. and Kane, M., The SHARE 709 System: Programming and Modification, *Journal of the ACM*, Vol. 6, No. 2, April 1959, pp. 128-133.

[GRIR70] Grishman, R., Criteria for a Debugging Language, *Debugging Techniques in Large Systems*, Proceedings Courant Computer Science Symposium 1, Rustin, R. (Ed.), June 29-July 1, 1970, pp. 50-75.

[GUPN84] Gupta, N. K. and Seviora, R. E., An Expert System Approach to Real-Time System Debugging, *Proceedings of the First Conference on Artificial Intelligence Applications*, 1984, pp. 336-343.

[HAMR81] Hamlet, R., Reliability Theory of Program Testing, *Acta Informatica*, Vol. 16, No. 1, August 1981, pp. 31-43.

[HANS85] Hanson, S. J. and Rosinski, R. R., Programmer Perceptions of Productivity and Programming Tools, *Communications of the ACM*, Vol. 28, No. 2, February 1985, pp. 180-189.

[HEEJ85] Heering, J. and Klint, P., Towards Monolingual Programming Environments, *ACM Transactions on Programming Languages and Systems*, Vol. 7, No. 2, April 1985, pp. 183-213.

[HELD85] Helmbold, D. and Luckham, D., Debugging Ada Tasking Programs, *IEEE Software*, Vol. 2, No. 2, March 1985, pp. 47-57.

[HENJ82] Hennessy, J., Symbolic Debugging of Optimized Code, *ACM Transactions on Programming Languages and Systems*, Vol. 4, No. 3, July 1982, pp. 323-344.

[HOAC69] Hoare, C. A. R., An Axiomatic Approach to Computer Programming, *Communications of the ACM*, Vol. 12, No. 10, October 1969, pp. 576-680.

[HOLD83] Holdsworth, D., A System for Analyzing ADA Programs at Run-time, *Software - Practice and Experience*, Vol. 13, No. 12, May 1983, pp. 407-421.

[HORR80] Horspool, R. N. and Marovac, N., An Approach to the Problem of Detranslation of Computer Programs, *The Computer Journal*, Vol. 23, No. 3, August 1980, pp. 223-229.

[HOWW76] Howden, W.E., Reliability of Path Analysis Testing Strategy, *IEEE Transaction on Software Engineering*, Vol. SE-2, No. 3, September 1976, pp. 208-214.

[HOWW78] Howden, W.E., An Evaluation of the Effectiveness of Symbolic Testing, *Software - Practice and Experience*, Vol. 8, No. 4, July/August 1978, pp. 381-397.

[HUAJ77] Huang, J.C., Error Detection Through Program Testing, *Current Trends In Programming Methodologies*, Vol. 2, R.T. Yeh (Ed.), Prentice-Hall, 1977, pp. 16-43.

[HUAJ80] Huang, J.C., Instrumenting Programs for Symbolic-Trace Generation, *IEEE Computer*, Vol. 13, No. 12, December 1980, pp. 17-23.

[IEEE83a] IEEE, Inc. [Eds.], *IEEE Standard Pascal Computer Programming Language*, Institute of Electrical and Electronic Engineers, New York, 1983, 128 pp.

[IEEE83b] IEEE, Inc. [Eds.], *IEEE Standard Glossary of Software Engineering Terminology*, Institute of Electrical and Electronic Engineers, New York, 1983.

[ISOS87] Isoda, S., Shimomura, T., and Ono, Y., VIPS: A Visual Debugger, *IEEE Software*, Vol. 4, No. 3, May 1987, pp. 8-19.

[JACJ74] Jachner, J. and Agarwal, V. K., Data Flow Anomaly Detection, *IEEE Transactions on Software Engineering*, Vol. SE-10, No. 4, July 1974 pp. 432-437.

[JACC84] Jacobi, C.P., Debugging Modula-2 Programs on the Lilith Computer, *Twenty-Eighth International IEEE Computer Society Conference*, San Francisco, CA., February 1984, pp. 442-445.

[JOHW85] Johnson, W. and Soloway, E., PROUST: Knowledge-Based Program Understanding, *IEEE Transactions on Software Engineering*, Vol. SE-11, No. 3, March 1985, pp. 267-275.

[KAHW82] Kahan, W and Coonen, J. T., The Near Orthogonality of Syntax, Semantics, and Diagnostics in Numerical Programming Environments, *The Relationship Between Numerical Computation and Programming Languages.* Conference Proceedings, August 1981, pp. 103-113.

[KAMS84] Kamin, S. and Archer, Myla, Partial Implementations of Abstract Data Types: A Dissenting View on Errors, *Semantics of Data Types*, Proceedings of an International Symposium, Lecture Notes in Computer Science, Vol. 173, June 1984, pp. 317-336.

[KATH79] Katself, H., *Sdb: A Symbolic Debugger*, Bell Labs, Holmdell, N.J., 1979.

[KNUD68] Knuth, D. E., Semantics of Context-Free Languages, *Mathematical Systems Theory*, Vol. 2, No. 2, 1968, pp. 127-145.

[KNUD81] Knuth, D. E., *The Art of Computer Programming. Volume 2: Seminumerical Algorithms*, (Second Edition), Addison-Wesley, 1981.

[KOTA61] Kotok, A., DEC Debugging Tape, *Memo MIT-1*, Massachusetts Institute of Technology, Cambridge, Mass., December 1961.

[LAFF84] Lafora, F., Reverse Execution in a Generalized Control Regime, *Computer Languages*, Vol. 9, No. 11, December 1984, pp. 183-192.

[LANP64] Landin, P. J., The Mechanical Evaluation of Expressions, *Computer Journal*, Vol. 6, No. 4, 1964, pp. 308-320.

[LAUP71] Lauer, P., Consistent and Complementary Formal Definitions of Programming Languages, Technical Report TR25-121, IBM Vienna Laboratory, November, 1971.

[LAUS79] Lausen, S., Debugging Techniques, *Software - Practice and Experience*, Vol. 9, No. 1, January 1979, pp. 51-63.

[LEEJ72] Lee, J. A. N., *Computer Semantics*, Van Nostrand Reinhold, New York, 1972.

[LEEP83] Lee, P. A., Exception Handling in C Programs, *Software - Practice and Experience*, Vol. 13, No. 5, May 1983, pp. 389-405.

[LOER76] Loeser, R. and Gaposchkin, E. M., The Second Law of Debugging, *Software - Practice and Experience*, Vol. 6, No. 4, October 1976, pp. 577-8.

[LUCP69] Lucas, P. and Walk, K., On the Formal Description of PL/1, *Annual Review of Automatic Programming*, Vol. 6, No. 3, 1969, pp. 105-82.

[LUKF80] Lukey, F. L., Understanding and Debugging Programs, *International Journal of Man-Machine Studies*, Vol. 12, No. 2, February 1980, pp. 189-202.

[LYLJ84] Lyle, J.R., Evaluating Variations on Program Slicing for Debugging, Ph. D. Dissertation, University of Maryland, 1984, 107 pp.

[MALJ83] Malone, J.R., Implementation of a Retrospective Tracing Facility, *Software - Practice and Experience*, Vol. 13, No. 9, September 1983, pp. 791-796.

[MANS84] Manno, S.J., Selectively Record your Microprocessor's History, *EDN*, Vol. 29, No. 19, September 1984, pp. 286-287.

[MCCJ63] McCarthy, J., Towards a Mathematical Science of Computation, *Information Processing 63*, Proceedings of the IFIP Congress 62, 1963, pp. 21-28.

[MCCJ66] McCarthy, J. A Formal Description of a Subset of Algol, *Proceedings of the IFIP Working Conference on Formal Language Description Language*, North Holland, 1966, pp. 1-12.

[MCGD80] McGregor, D. R. and Malone, J. R., STABDUMP - A Dump Interpreter to Assist Debugging, *Software - Practice and Experience*, Vol. 10, No. 4, April 1980, pp. 329-332.

[MILB88] Miller, B. and Choi, J.D., A Mechanism for Efficient Debugging of Parallel Programs, *Proceedings of the SIGPLAN '88 Conference on Program Language Design and Implementation*, published in *SIGPLAN Notices* Vol. 23, No. 7, July 1988, pp. 135-144.

[MINM67] Minsky, M., *Computation: Finite and Infinite Machines*, Prentice-Hall, 1967.

[MURD85] Murray, D., Resourceful Debugger Sifts Through Faults in IEEE-802.3 LANs, *AFIPS National Computer Conference*, Vol. 33, No. 2, 24 January 1985, pp. 173-80.

[MYEG81] Myers, G. J. and Hocker, David G., The Use of Software Simulators in the Testing and Debugging of Microprogram Logic, *IEEE Transactions on Computers*, Vol. c-30, No. 7, July 1981, pp. 519-523.

[MYEG79] Myers, G. J., *The Art of Software Testing*, 1979.

[NASI73] Nassi, I., and Schneiderman, B., Flowchart Techniques for Structured Programming, *ACM SIGPLAN Notices*, Vol. 8., No. 8, August 1973, pp. 12-26.

[NAUP69] Naur, P. and Randell, B. (Eds.) *Software Engineering*, NATO Scientific Affairs Division, Brussels, Belgium, 1969.

[OSTL76] Osterweil, L.J. and Fosdick, L.D., DAVE - A Validation Error Detection and Documentation System for FORTRAN programmers, *Software - Practice and Experience*, Vol. 6, No. 4, October/December 1976, pp. 473-486.

[OSTL81] Osterweil, L.J., Fosdick, L.D., and Taylor, R.N., Error and Anomaly Diagnosis through Data Flow Analysis, *Computer Program Testing*, Chandrasekaran and Radicchi (Eds.), North Holland, 1981.

[OSTT84] Ostrand, T. J. and Weyuker, Elain J., Collecting and Categorizing Software Error Data in an Industrial Environment, *The Journal of Systems and Software*, Vol. 4, No. 4, November 1984, pp. 289-300.

[PAGF78] Pagan, F. G., Formal Semantics of a SNOBOL4 Subset, *Computer Languages*, Vol. 3, No. 1, 1978, pp.13-30.

[PANS85] Panchapakesan, S., Subramanian, S.S., and Venkateswaran, H., An Interactive Assembly Level Debugger, *Software - Practice and Experience*, Vol. 15, No. 1, January 1985, pp. 59-64.

[PIER74] Pierce, R. H., Source Language Debugging on a Small Computer, *The Computer Journal*, Vol. 17, No. 4, November 1974, pp. 313-317.

[PLAB81] Plattner, B. and Nievergelt, J., Monitoring Program Execution: A Survey, *IEEE Computer*, Vol. 14, No. 11, November 1981, pp. 76-93.

[PLAB84] Plattner, B., Real-Time Execution Monitoring, *IEEE Transactions on Software Engineering*, Vol. SE-10, No. 6, November 1984, pp. 756-764.

[RAMC75] Ramamoorthy, C.V., Kim, K.H., and Kim, W.T., Optimal Placement of Software Monitors Aiding Systematic Testing, *IEEE Transaction on Software Engineering*, Vol. SE-1, No. 4, December 1975, pp. 403-410.

[REIS84] Reiss, S.P., Graphical Program Development with PECAN Program Development Systems, *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, Pittsburgh, PA, April 1984, pp. 31-41.

[REIS85] Reiss, S.P., PECAN: Program Development Systems that Support Multiple Views, *IEEE Transactions on Software Engineering*, Vol. SE-11, No. 3, March 1985, pp. 276-285.

[RITD78] Ritchie, D.M., and Thompson, K., The UNIX Time-Sharing System, *The Bell System Technical Journal*, Vol. 57, No. 6, Part 2, July/August 1978, pp. 1905-1930.

[ROGH67] Rogers, H., *Theory of Recursive Functions and Effective Computability*, McGraw-Hill, 1967.

[SATE79] Satterthwaite, E.H., *Source Language Debugging Tools*, Garland Publications, 1979.

[SCHJ70] Schwartz, J. T., An Overview of Bugs, *Debugging Techniques in Large Systems*, Proceedings Courant Computer Science Symposium 1, Rustin, R. (Ed.), June 29-July 1, 1970, pp. 1-16.

[SEVR87] Seviora, R. E., Knowledge-Based Program Debugging Systems, *IEEE Software*, Vol. 4, No. 3, May 1987, pp. 20-32.

[SHAE82] Shapiro, E.Y., *Algorithmic Program Debugging*, ACM Distinguished Dissertations, MIT Press, 1982, 232 pp.

[STUL73] Stucky, L.G., Automatic Generation fo Self-Metric Software, *Proceedings of the 1973 IEEE Symposium on Computer Software Reliability*, IEEE Computer Society, April 1973, pp. 94-100.

[SMIE85] Smith, E. T., A Debugger for Message-based Processes, *Software - Practice and Experience*, Vol. 15, No. 11, November 1985, pp. 1073-1086.

[STEJ84] Steffen, J. L., Experience with a Portable Debugging Tool, *Software - Practice and Experience*, Vol. 14, No. 4, April 1984, pp. 323-334.

[TEIW85] Teitelman, W., A Tour Through Cedar, *IEEE Transactions on Software Engineering*, Vol. SE-11, No. 3, March 1985, pp. 285-301.

[TENR76] Tennent, R. D., The Denotational Semantics of Programming Languages, *Communications of the ACM*, Vol. 19, No. 8, August 1976, pp. 437-453.

[TOLS84] Tolchin, S.G., Bergan, E.S., Espenshade, M.A., Grossman, R.S., Schneider, M.J., Sterne, D.F., and Wacther, R.F., An Expert System for the Manipulation of Programs, *4th Jerusalem Conf. of Information Technology*, May 1984, pp. 460-467.

[TOUS84] Toueg, S., Babaoglu, O., On the Optimum Checkpoint Selection Problem, *SIAM Journal of Computing*, Vol. 13, No. 3, August 1984, pp. 630-649.

[TRAM79] Tratner, M., A Fundamental Approach to Debugging, *Software - Practice and Experience*, Vol. 9, No. 2, February 1979, pp. 97-99.

[TSUH86] Tsubotani, H., Monden, N., Tanaka, M., and Ichikawa, T., A High Level Language-Based Computing Environment to Support Production and Execution of Reliable Programs, *IEEE Transactions on Software Engineering*, Vol. SE-12, No. 2, February 1986, pp. 134-146.

[TURA36] Turing, A. M., On Computable Numbers with an Application to the Entscheidungsproblem, *Proceedings of the London Mathematical Society*, Ser. 2-42, 1936, pp. 230-265.

[VESI86] Vessey, I., Expertise in Debugging Computer Programs: An Analysis of the Content of Verbal Protocols, *IEEE Transactions on Systems, Man, and Cybernetics*, Vol. SMC-16, No. 5, September/October 1986, pp. 621-637.

[VOGU80] Voges, U., Gmeiner, L., and von Mayrhauser, A.A., SADAT -- An Automated Testing Tool, *IEEE Transactions on Software Engineering*, Vol. SE-6, No. 3, May 1980, pp. 286-290.

[WATD79] Watt, D. A., An Extended Attribute Grammar for Pascal, *SIGPLAN Notices*, Vol. 14, No. 2, February 1979, pp. 60-74.

[WATD81] Watt, D. A. and Findlay, W., A Pascal diagnosis system, in *Pascal - the Language and its Implementation*, D. W. Barron (Ed.), 1981.

[WATD83] Watt, D. A. and Madsen, O. L., Extended Attribute Grammars, *The Computer Journal*, Vol. 26, No. 2, May 1983, pp. 142-153.

[WEGP72] Wegner, P., The Vienna Definition Language, *Computing Surveys*, Vol. 4, No. 1, March 1972, pp. 5-63.

[WEIM82] Weiser, M., Programmers Use Slices When Debugging, *Communications of the ACM*, Vol. 25, No. 7, July 1982, pp. 446-452.

[WEIM84] Weiser, M. Program Slicing, *IEEE Transaction on Software Engineering*, Vol. SE-16, No. 4, July 1984, pp. 352-357.

[WELJ77] Welch, J., Sneeringer, W., and Hoare, C.A.R., "Ambiguities and Insecurities in Pascal", *Software - Practice and Experience*, Vol. 7, No. 6, l977, pp.685-696.

[WERH82] Wertz, H., Stereotyped Program Debugging: An Aid for Novice Programmers, *International Journal of Man Machine Studies*, Vol. 16, No. 4, May 1982, pp. 379-392.

[WHIN85] White, N. H. and Bennett, K. H., PRTDS : - A Pascal Run-Time Diagnostics System, *Software - Practice and Experience*, Vol. 15, No. 11, November 1985, pp. 1041-1056.

[WILM57] Wilkes, M., Wheeler, D.J. and Gill, S., The Preparation of Programs for and Electronic Digital Computer, (2nd edition), Addison-Wesley, 1957, 238 pp.

[WIRN71] Wirth, N., The Programming Language Pascal, *Acta Informatica*, Vol. 1, No. 1, 1971, pp.35-63.

[WITC83] Witschorik, C. A., The Real: Time Debugging Monitor for the Bell System 1A Processor, *Software - Practice and Experience*, Vol. 13, No. 8, August 1983, pp. 272-743.

[YEMS85] Yemini, S. and Berry, Daniel M., A Modular Verifiable Exception: Handling Mechanism, *ACM Transactions on Programming Languages and Systems*, Vol. 7, No. 2, April 1985, pp. 214-243.

# Appendix A. An Example Attribute Grammar

This appendix gives an attribute grammar [KNUD68] for the abstract syntax of a simplified Level 1 language. As discussed in Chapters 3 and 4, this grammar is designed to calculate the statement attributes needed to model flow of control. The required attributes are NUMBER, which gives a unique sequence number to each assignment statement and to each test, and TD and FD, which provide true and false destinations for each test. (A test is the conditional part of an if or while statement.) This appendix discusses first the unattributed grammar, then the attributes associated with each nonterminal and terminal symbol, and finally the calculation of the values of the attributes. Completing the appendix is a listing of the unattributed grammar, the attributes, and the attributed grammar. In the listing and in the discussion, nonterminal symbols are enclosed within angle brackets, terminals are written in bold, and values taken by attributes are represented by expressions involving values written in lower case.

The unattributed grammar shows that a program contains a sequence of statements each of which is an assignment, an **if**, or a **while** statement. If and while statements are compound and respectively have two or one embedded statement sequences. The terminal symbol **assign** represents the assignment statement, and the symbol **test** represents the conditional part of an if or while statement. No variables, constants or expressions are included in the simplified language.

Each nonterminal has two inherited attributes, ↓START and ↓NEXT, and a single synthesized attribute, ↑STOP. Attribute ↓START is the number of the first (or only) statement derived from that nonterminal and similarly, ↑STOP is the number of the last (or only) statement derived. ↓NEXT is the number of the statement that is to be executed after the last statement derived from the nonterminal, i.e. the last statement's logical successor. The terminal symbol **assign** has two inherited attributes: ↓NUMBER is the sequence number of the assignment and ↓NEXT is the sequence number of the statement that is the logical successor to the assignment. Similarly, terminal symbol **test** has three inherited attributes: ↓NUMBER is the sequence number of the test, and ↓TD and ↓FD represent the sequence numbers of the statements to execute if the test is true or false, respectively. Attributes ↓TD and ↓FD then are the true and false destinations of the test. These inherited attributes of the terminal symbols thus provide the information needed to model control flow. The attributes of the nonterminals aid in the calculation of the required attributes. This calculation is discussed below.

In the attributed grammar, the first production group defines the inherited attributes for the statement list that constitutes the program. This list begins with statement 1 and thus the ↓START attribute position has the value ↓1. The successor of the last statement in the program is not defined and thus the ↓NEXT attribute has the value ↓UNDEFINED. The attribute value ↑end is the synthesized number of the last statement in the program.

In the first option of the second production group, the ↓START attribute of <stmt> and the ↓NEXT attribute of <stmtlist$_2$> come respectively from the

$\downarrow$START and $\downarrow$NEXT attributes of <stmtlist$_1$>. The attribute value $\uparrow$end$_1$ is the number of the last statement derived from <stmt> and thus <stmt>'s $\downarrow$NEXT attribute is $\uparrow$end$_1$ + 1, and <stmtlist$_2$>'s $\downarrow$START attribute has the value $\downarrow$end$_1$ + 1. The attribute value $\uparrow$end$_2$ is the number of the last statement derived from <stmtlist$_2$>; it provides <stmtlist$_1$>'s $\uparrow$STOP attribute.

The second option of the second production group is simpler than the first. The inherited values of the $\downarrow$START and $\downarrow$NEXT attributes for <stmtlist$_1$> provide the values for the corresponding attributes of <stmt>, and the synthesized value $\uparrow$end$_2$ provides the value for the $\uparrow$STOP attribute of <stmtlist$_1$>. Similarly, in the third production group, the inherited values on the right side of the productions take their values from corresponding attributes on the left side, and the synthesized values on the left side take values from corresponding attributes on the right.

In the fourth group, the sequence number of the test takes its value from the $\downarrow$START attribute of the if statement, while the value of the $\downarrow$TD attribute of **test** is $\downarrow$begin + 1, i.e. the sequence number of the first statement of the true branch, and the value of the $\downarrow$FD attribute of the test is $\downarrow$end$_1$ + 1. The value $\downarrow$end$_1$ is of course the sequence number of the last statement in the true branch of the if statement, and thus $\downarrow$end$_1$ + 1 is the sequence number of the first statement of the false branch. Since control transfers from the end of either branch to the logical successor of the if, then the $\downarrow$NEXT attribute of <ifthenelse> and of the two embedded statement lists are all identical.

For the while statement (i.e. production group 5), the $\downarrow$TD attribute of the test is $\downarrow$begin + 1, i.e. the sequence number of the first statement of the embedded statement list and the $\downarrow$FD attribute of the test is $\downarrow$successor, the $\downarrow$NEXT attribute of the while statement, i.e. the while statement's successor. Also, the $\downarrow$NEXT attribute of the embedded sequence <stmtlist> is $\downarrow$begin, the sequence number of the while statement's test, because the test must be executed following each execution of the last statement in the loop.

Finally, in the assignment statement shown in group 6, the ↓NUMBER and ↓NEXT attributes are taken directly from the <assignment> nonterminal. This completes the discussion of the attributes, and the following unattributed grammar, attribute listing, and attributed grammar complete the appendix.

## Unattributed Grammar

```
<program> ::= <stmtlist>
<stmtlist> ::= <stmt> <stmtlist>
             | <stmt>

<stmt> ::= <ifthenelse>
         | <whileloop>
         | <assignment>

<ifthenelse> ::= test <stmtlist> <stmtlist>

<whileloop> ::= test <stmtlist>

<assignment> ::= assign
```

## Attributes of Nonterminals

<stmtlist> ↓START ↑STOP ↓NEXT
<stmt> ↓START ↑STOP ↓NEXT

<ifthenelse> ↓START ↑STOP ↓NEXT
<whileloop> ↓START ↑STOP ↓NEXT

<assignment> ↓START ↑STOP ↓NEXT

## Attributes of Terminals

**assign** ↓NUMBER ↓NEXT
**test** ↓NUMBER ↓TD ↓FD

| <u>Attributed Grammar</u> | **Production Group** |
|---|---|
| <program> ::= <stmtlist> $\downarrow 1$ $\uparrow$end $\downarrow$*UNDEFINED* | 1 |
| <stmtlist$_1$> $\downarrow$begin $\uparrow$end$_2$ $\downarrow$successor ::= <stmt> $\downarrow$begin $\uparrow$end$_1$ $\downarrow$end$_1$+1 <stmtlist$_2$> $\downarrow$end$_1$+1 $\uparrow$end$_2$ $\downarrow$successor | 2 |
| \| <stmt> $\downarrow$begin $\uparrow$end$_2$ $\downarrow$successor | |
| <stmt> $\downarrow$begin $\uparrow$end $\downarrow$successor ::= <ifthenelse> $\downarrow$begin $\uparrow$end $\downarrow$successor | 3 |
| \| <whileloop> $\downarrow$begin $\uparrow$end $\downarrow$successor | |
| \| <assignment> $\downarrow$begin $\uparrow$end $\downarrow$successor | |
| <ifthenelse> $\downarrow$begin $\uparrow$end $\downarrow$successor ::= **test** $\downarrow$begin $\downarrow$begin+1 $\downarrow$end$_1$+1 <stmtlist> $\downarrow$begin+1 $\uparrow$end$_1$ $\downarrow$successor <stmtlist> $\downarrow$end$_1$+1 $\uparrow$end $\downarrow$successor | 4 |
| <whileloop> $\downarrow$begin $\uparrow$end $\downarrow$successor ::= **test** $\downarrow$begin $\downarrow$begin+1 $\downarrow$successor <stmtlist> $\downarrow$begin+1 $\uparrow$end $\downarrow$begin | 5 |
| <assignment> $\downarrow$begin $\uparrow$begin $\downarrow$successor ::= **assign** $\downarrow$begin $\downarrow$successor | 6 |

# Index

The vita has been removed from the scanned document