Two Dimensional Pinpointing: An Application of Formal Specification to Debugging Packages

David Luckham

Sriram Sankar

Shuzo Takahashi

Technical Report No. CSL-TR-89-379

Program Analysis and Verification Group Report No. 40

April 1989

This research was supported by the Defense Advanced Research Projects Agency under contract N00039-84-C-0211.

Two Dimensional Pinpointing: An Application of Formal Specification to Debugging Packages*

David Luckham[†] Sriram Sankar[†] Shuzo Takahashi[‡]

April

Computer Systems Laboratory Technical Report CSL-TR-89-379
Program Analysis and Verification Group Report No. 40

Abstract

New methods of testing and debugging software utilizing high-level formal specifications are presented. These methods require a new generation of support tools. Such tools must be capable of automatically comparing the runtime behavior of hierarchically structured software with high-level specifications; they must provide information about inconsistencies in terms of abstractions used in specifications.

This use of specifications has several advantages over present-day debugging methods: (1) the debugging problem itself is precisely defined by specifications; (2) violations of specifications are detected automatically, thus eliminating the need to search output traces and recognize errors manually; (3) complex tests, such as tests for side-effects on global data, can be made easily; (4) the new methods are independent of any compiler and runtime environment for a programming language; (5) they apply generally to hierarchically structured software — e.g., packages containing nested units, (6) they also apply to other life-cycle processes such as analysis of prototypes, and the use of prototypes to build formal specifications.

In this paper a particular process for locating errors in software packages, called ***two dimensional pinpointing,*** is described. Tests consist of sequences of package operations (first dimension). Specifications at the highest (most abstract) level are checked first. If violations occur then new specifications are added if possible, otherwise checking of specifications at the next lower level (second dimension) is activated. Violation of a new specification provides more information about the error which reduces the region of program text under suspicion. All interaction between programmer and toolset is phrased in terms of the concepts used to specify the program.

Two dimensional pinpointing is presented using the Anna specification language for Ada programs. Anna and a toolset for comparing behavior of Ada programs with Anna specifications is described. Pinpointing techniques are then illustrated by examples. The examples involve debugging of Ada packages, for which Anna provides a rich set of specification constructs. The Anna toolset supports use of the methodology on the full Ada/Anna languages, and is being engineered to commercial standards.

Keywords: Ada, Anna, Data Abstraction, Debugging, Package, Specification, Testing, Verification

*This research was supported by the Defense Advanced Research Projects Agency under contract N00039-84-C-0211.

[†]Program Analysis and Verification Group, Computer Systems Laboratory, Stanford University, Stanford, California 94305.

[‡]Odyssey Research Associates, 525 Middlefield Road, Suite 250, Menlo Park, California 94025.

1 Introduction

Specification languages present an opportunity to develop new techniques in all phases of software production. Essentially, these languages provide facilities for expressing information about programs that is not normally part of the text of a program. Specifications are expressed in a machine processable form — they can be parsed, checked **for static semantic** errors, and in many cases, compiled into runtime tests. Consequently, they are called ***formal specifications***[1]. Typical examples of specification languages developed during the past few years are Anna [6], Larch [9] and RAISE [15].

Debugging has always been a creative endeavor. A great deal of ingenuity and effort has gone into using the current generation of rather crude debugging tools to discover errors in programs. The main point of this paper is that much more powerful debugging techniques can be developed based on formal specifications. Our new techniques utilize both the high level concepts used in specifications, and the abstraction and information-hiding constructs used in modern programming languages.

There are two aspects to developing new software production techniques.

1. ***developing a new capability.***

   In our case, we have designed the Anna specification language for Ada programs, and developed a suite of tools for checking the runtime behavior of programs for consistency with specifications [11,17]. These tools are collectively called ***the Annalyaer.***

2. ***defining methods of applying the capability.***

   The Annalyzer can be used to develop adequate formal specifications from prototypes. This is done by checking a proposed (but insufficient) specification against behavior of various prototypes of the software — as we show later, inconsistencies can be used to find missing specifications. Alternatively, given an accepted specification, the Annalyzer can be used to test and debug software.

   In this paper we describe some techniques for pinpointing errors in hierarchically structured software using formal specifications and the Annalyzer.

We present a view of debugging as an activity that starts with a specification and an executable program. Both have been constructed during earlier activities in the software development process. The goal is to establish that a program behaves consistently with its specifications.

We suppose that the program is a complex unit such as an Ada package, with several structural levels and nested units which may share common data. Each level has specifications. Typically, the top level specifications express ***what*** the unit itself does using high-level abstract concepts. Lower level specifications express what the nested units do, and how they are related, by referring to implementation details that are hidden at higher program levels.

In general, two strategies are involved in testing a package. First, tests involve executing sequences of package operations and using the Annalyzer to compare their runtime behavior with the specifications — called ***sequence testing?*** Second, only the highest level specifications are checked unless a violation occurs.

[1]Henceforth ***we*** use ***specification*** to mean ***formal specification.***

[2]The reason for using sequences of operations is discussed later.

Thus, we regard the debugging problem as having two dimensions: the length of the test sequence, and the structural levels of the package.

Two dimensional pinpointing starts if and when inconsistencies arise. When an inconsistency arises, a ***region of suspicion*** is defined. It is a region of a package, including both specifications and Ada text, that contains an inconsistency on the test sequence. It may include not only the (sub) unit whose execution propagated the inconsistency, but also other units at the same level that are related to it, or preceded it in the test sequence. The goal of pinpointing is to reduce the region in which inconsistent behavior can be detected to a single declarative region or subprogram.

Pinpointing strategies utilize the hierarchical structure of the program. New, more detailed specifications and ***annotations***[3] about program behavior, are added to the previous specifications — this is called ***augmenting*** specifications. New specifications are first added at the same level as the one that was violated. They should be related to the violated one in such a way that they are likely to be violated by the same test[4]. The same test sequence is then repeated. A violation of a new specification may reduce the region of suspicion by providing more information, and possibly by occurring earlier in the test sequence. ***At any one time only a few high level specifications are checked on a given test sequence.***

When the region of suspicion has been reduced as much as possible by augmenting specifications at one level, the next lower level is considered. At this new level, checking of specifications and annotations of those nested units that remain in the region of suspicion is activated. This strategy is repeated at progressively lower levels of the program. Each time a violation occurs, the length of the test sequence is shorter, or the level of the violated specification is lower — i.e., one of the two dimensions of the test is reduced.

Repair involves changes to executable code or data structures in a single declarative region or subprogram body.

This hierarchical use of specifications may be employed with a range of informal or formal methods. For example, in pinpointing, the methods whereby new specifications and annotations are created may be highly intuitive, such as "guess-and-test". Alternatively, new specifications may be formally proved to imply the violated ones before they are tested. When repairs are made, goal-oriented techniques utilizing the specifications and annotations can be used [8]. Or repairs can be proved consistent with specifications that were previously violated.

Advantages of debugging with specifications over present debugging methods and tools include:

- ***The debugging problem is precisely defined.***
  The set of specifications to be tested constitute an accurate formal definition of the behavior to be tested. If, later on, new specifications are required, analysis of their relationship to the old specifications will indicate what further tests need to be performed.

- ***Violations of specifications are detected automatically.***
  The task of searching output traces in order to recognize errors is eliminated.

[3]Specifications applying over small local scopes are called annotations.

[4]Guidelines on how to do this are described in examples later.

- ***Violations can be analyzed for their effect on users of the software.***
  Violation of a high level specification in the interface of a module or package indicates explicitly which facilities are unreliable. Users of other facilities may be unaffected.

- ***During debugging, new specifications can be expressed formally at any program level, and then tested by the same methods and tools.***
  A user interacts with the Annalyzer by formulating new specifications and submitting them to tests. He no longer has to deduce whether an abstract property of the program is violated from more primitive data.

- ***Very complex tests can be formulated easily and checked automatically.***
  For example, specifications against side-effects on global data are easily formalized and tested. This is difficult to do with standard debuggers.

- ***Our methods and tools can be used with*** any ***Ada compiler and runtime implementation.***

- ***Our methods apply equally well to debugging other kinds of hierarchically structured software modules and to concurrent programs.***
  Indeed, it is in the area of testing and debugging complex software that the advantages of formal specifications become obvious. For concurrent software, the methods use specification languages that extend Anna by providing new constructs for concurrent behavior — e.g., TSL [5].

In this paper we describe guidelines and techniques of 2-dimensional pinpointing. Our examples show the methodology applied to an Ada package with Anna specifications[5].

The paper is structured as follows. Section 2 presents an overview of the specification language Anna. Section 3 describes the Anna consistency checking system as it appears to a user. Section 4 presents an outline of the general methodology of 2-dimensional pinpointing.

Section 5 presents examples and guidelines. We show part of an interactive session with the Annalyzer to test and debug an Ada package. Although the software and specifications in this example are simple, it contains some complexity that may be encountered in real applications packages. Basic pinpointing techniques are shown, such as augmenting specifications to reduce the region in which code repairs are considered. Both high level specifications and low level local annotations are used. As will be seen, all interactions take place using specification concepts of various levels of the package — i.e., each scope level of the package has a set of concepts used to specify that level, and pinpointing at a particular level uses only those concepts.

This paper is a short introduction to debugging with specifications. Many topics, such as the use of algebraic specifications, are omitted. A fuller treatment is given in [12]. There we describe techniques of constructing specifications so as to justify the postulate that debugging starts with ***correct*** specifications. More detailed examples of pinpointing are given, and the reasoning to justify various steps is discussed in detail, some of it formally.

[5]The paper is self-contained, and prior knowledge of Anna is not necessary; some familiarity with Ada or other Algol-like languages such as Modula-2 is assumed.

Relationship with formal methods of program development. Formal methods may be divided into two categories, strict and relaxed. Strict formal methods result in programs that provably possess some property. Program Development by transformation of specifications, and Program Verification by proving consistency between specification and program, are two examples of strict methods. Historically, strict methods and their support tools have lagged behind the current generation of programs and programming languages, while demanding advanced knowledge beyond the training of current programmers.

Relaxed methods utilize only parts of the formal basis of strict methods, but are easier to use, and can be applied to wider classes of programs. They improve upon current informal methods, and increase the likelihood of developing programs that possess certain properties — but not always provably so.

The hierarchical debugging methods we describe here are relaxed formal methods. They require using a specification language to express the concepts upon which a program is based, the functional goals of the program, and the subgoals of its various parts. Also, informal reasoning, which all programmers use now and then, is applied to formal specifications and their program contexts. However, we stop short of requiring the use of mathematical proof to justify various steps. Our first goal is to encourage the application of specifications to programming problems. Eventually, someone who catches on to using specifications to debug, may well use rigorous formal proofs in place of informal reasoning, because the information required for such proofs has already been expressed explicitly in specifications and annot ations.

Utilizing specifications in debugging may be viewed as an evolutionary step towards formalizing every stage of software development. Most of the methodologies being researched at present — e.g., VDM [3], program transformations [2] — propose a top-down process whereby abstract specifications are developed first ***(design*** and ***specification),*** and refined in a series of detailed steps resulting in executable software. If these methodologies are successful eventually, debugging will be unnecessary. However, it is generally admitted that these methodologies, when applied, are not strictly top-down. In other words, activities similar to those that constitute debugging also take place during earlier stages. Thus, the kinds of methods described here also apply earlier in the development process — e.g., in analysis for missing specifications.

The role of formal proof in software development has been the subject of much research in Program Verification. A current major issue is how best to integrate proof into the process of building software — see [4], [8] and [19]. Here, we have chosen to rely on informal reasoning as a method of making diagnostic decisions, and justifying repairs. Certainly, formal proof can be substituted for informal reasoning any place we use it. If our methods turn out to be useful, they will indicate ways to integrate formal proof into the process. This will be appropriate when automated proof tools mature to a point where they provide adequate support.

Relationship with current debugging methods. In this paper we do not address the problem of test data selection, which is the subject of much current literature (see e.g., [7,10,20]). Neither do we deal here with the problem of automatically generating test sequences from formal specifications?. However, we do postulate the need for sequence testing of complex modules such as Ada packages. Our methods are set up to deal with general tests of this kind.

Essentially, the Annalyzer is an ***Oracle*** which judges the program behavior on tests[7]. Having actually implemented an Oracle, we deal with the problem of how best to use it on a single sequence test which so happens to uncover an inconsistency.

Current debuggers gather information from the lowest implementation levels of a program and its runtime environment. The programmer must deduce what is happening from this information. Some more advanced debuggers will test boolean assertions about program variables. The Annalyzer, on the other hand, provides the power to test very general constraints on packages, abstract types, data structures, and subprograms, as explained in the next Section. So the Annalyzer may be viewed as an evolutionary step in the development of debugging tools, matching the evolution of programming languages in abstraction and structuring constructs.

2 An Overview of Anna and the Checking Methodology

Anna (ANNotated Ada) is a language extension of Ada [1] to include facilities for formally specifying the intended behavior of Ada programs. Anna was designed to meet a perceived need to augment Ada with precise machine-processable annotations so that well established formal methods of specification and documentation can be applied to Ada programs. In this section we give a brief outline of a few kinds of annotations. A complete definition of Anna is given in [13].

Anna is based on first-order logic and its syntax is a straightforward extension of the Ada syntax. Anna constructs appear as ***formal comments*** within the Ada source text (within the Ada comment framework). Anna defines two kinds of formal comments, which are introduced by special comment indicators in order to distinguish them from informal comments. These formal comments are ***virtual Ada text,*** each line of which begins with the indicator `--:`, and ***annotations,*** each line of which begins with the indicator `--|`.

In the discussion that follows, an overview of the checking methodology[8] is given corresponding to each of the Anna constructs discussed. Details of the checking methodology can be found in [16,17,18].

2.1 Virtual Ada Text

Virtual Ada text is Ada text appearing as formal comments, but otherwise obeying all of the Ada language rules. Virtual text may refer to actual text, but is not allowed to affect the computation of the actual program. Actual text cannot refer to virtual text. The purpose of virtual Ada text is to define ***concepts***[9] used in annotations. Often the formal specifications of a program will refer to concepts that are not explicitly implemented as part

[6]This is a promising research direction, which might be amenable to guidelines similar to **the ones we** give for pinpointing.

[7]A hypothetical Oracle is an **a priori** assumption in much testing methodology.

[8]The checking methodology refers to how annotations are checked at runtime.

[9]Functions used in annotations are called concepts.

of the program. These concepts can be defined as virtual Ada text declarations. Virtual Ada text may also be used to ***compute*** values that are not computed by the actual program, but that are useful in defining the behavior of the program.

Example** of **virtual text:

```
    package QUEUE-MANAGER is
        ...
        type QUEUE is private;
        ...
--:     function IS_MEMBER(E : ELEMENT; Q : QUEUE) return BOOLEAN;
        ...
    end QUEUE-MANAGER;
```

In the above example, IS-MEMBER is a virtual function. It is not an actual operation of the package QUEUE-MANAGER. It is used in annotations of actual subprograms of QUEUE-MANAGER.

2.2 Annotations

Annotations are constraints on the underlying Ada program. They are made up of expressions that are boolean-valued. The location of an annotation in the Ada program together with its syntactic structure indicates the kind of constraints that the annotation imposes on the underlying program. Anna provides different kinds of annotations, each associated with a particular Ada construct. The Anna expressions extend (i.e., are a superset of the expressions in Ada. The kinds of annotations used in the example of Section 5 are described below. Along with each description is a paragraph describing how the Annalyzer checks for consistency against the corresponding Anna construct. Note that there are many more constructs in Anna that are not explained below since their explanation is not necessary for the understanding of the example.

Type Annotations: A type or subtype annotation is a constraint on an Ada type. The constraint applies throughout the scope of the type definition. Type annotations are located immediately after the definition of the type they constrain, and are bound to the type definition by the keyword **where.**

Example** of **a type annotation:

```
    type QUEUE is
        record
            STORE : QUEUE_ARRAY(1. . MAX);
            IN-PTR, OUT-PTR: INTEGER range 1. . MAX := 1;
            SIZE: INTEGER range 0. . MAX := 0;
        end record;
--| where Q:QUEUE =>
--|     (Q . IN_PTR - Q . OUT-PTR - Q . SIZE) mod MAX = 0;
```

The above type annotation constrains all values of the type **QUEUE** so that their components, **IN-PTR, OUT-PTR** and **SIZE** satisfy the equation in the annotation.

> ***Checking methodology:*** Subtype annotations are converted into functions that test for the truth of the annotations. Calls to these functions are inserted at all places where values of the type are generated. Examples of such places are assignment statements and type conversions.

In the example of Section 5, this annotation occurs in a modified form:

```
--| where in out Q : QUEUE =>
--|     (Q. IN-PTR - Q . OUT-PTR - Q . SIZE) mod MAX = 0;
```

This is a ***modified type annotation.*** Modified type annotations can be used only in packages. The **in out** indicates that values of the type are constrained only at the beginning and end of each package operation.

> ***Checking methodology:*** Modified annotations are checked in a similar manner to subprogram annotations. See the discussion of subprogram annotations below.

Subprogram Annotations: ***Subprogram annotations*** are used to describe the behavior of subprograms. They are bound to an Ada subprogram specification by the keyword where. Subprogram declarations are annotated by lists of annotations. In general, each annotation of a subprogram specifies an input condition, an output condition, or an exceptional condition. Also subprogram annotations of functions may specify the resulting value returned by a function.

Examples** of **subprogram annotations:

```
    function IS_FULL(Q : QUEUE) return BOOLEAN;
  | <<SPEC_IS_FULL>>
  | where
  |     return LENGTH(Q) = MAX;

    procedure INSERT(E : ELEMENT; Q : in out QUEUE);
--| <<SPEC_INSERT>>
--| where
--|     out(LENGTH(Q) = LENGTH(in Q) + 1),
--|     out(IS_MEMBER(E, Q));
```

The first of the above examples is that of a result annotation. It specifies that the value returned by the function **IS-FULL** is the boolean value **LENGTH(Q)** = **MAX.** The annotations on the procedure **INSERT** are ***out*** annotations. ***Out*** annotations of **INSERT** must be satisfied whenever a call to **INSERT** terminates normally — i.e., terminates without propagating an exception. Whenever a call terminates normally, the length of the resulting Q must be ***one*** more than the length of the value of Q on entry to **INSERT.** Also, on termination, **E** must be a member of Q.

The above two examples have an extra feature — annotation names. Annotation names are given using Ada's label syntax. They are useful in referring to the annotations from the Annalyzer .

More detailed annotations called ***statement annotations*** can appear in the body of the subprogram. These annotations can be used to specify details of the subprogram at the statement level.

> ***Checking methodology:*** Explicit checks are inserted for result annotations immediately before every ***return*** statement in its scope. ***In*** and ***out*** annotations of subprograms are checked by explicit checks at the beginning of subprograms and at all possible exit points in the subprogram.

Exception Annotations: ***Exception annotations*** (or propagation annotations) specify the exceptional behavior of program units. There are two different kinds of exception annotations — strong propagation annotations and weak propagation annotations.

A ***strong propagation annotation*** specifies conditions under which exceptions should be propagated. The conditions are with respect to the initial state of the scope of the annotation. If the conditions are satisfied, then the scope of the annotation must be terminated by propagating the specified exception.

Example of a strong propagation annotation:

```
     procedure INSERT(E : ELEMENT; Q : in out QUEUE);
--| where
--       IS-FULL(Q) => raise FULL;
```

This annotations specifies that if **IS-FULL(Q)** is true on entry to **INSERT,** then **INSERT** must terminate by propagating the exception **FULL.**

A ***weak propagation annotation*** specifies what happens when an exception is propagated. It specifies conditions that must be satisfied if the scope of the annotation is terminated by propagating one or more specified exceptions.

Example of a weak propagation annotation:

```
     procedure INSERT( E : ELEMENT; Q : in out QUEUE);
--| where
--|      raise FULL => Q = in Q;
```

This annotation specifies that if the procedure **INSERT** terminates by propagating the exception **FULL,** then **INSERT** does not change the value of Q.

> ***Checking methodology:*** Exception annotations are checked by enclosing their scope by an Ada block statement. Exception handlers and ***out*** annotations are inserted into this block to ensure that all normal and abnormal terminations from the scope of the exception annotations satisfy the conditions imposed by these annotations.

Note that ***out*** annotations do not specify abnormal termination.

3 The Annalyzer Tool Suite

The Annalyzer tool suite is a set of programs that convert Anna formal comments into run-time checking code. This checking code is inserted into the underlying Ada program. When the resulting Ada program is executed, the checking code ensures that any inconsistency in the program with respect to the annotations is detected and reported. The resulting Ada program is linked to a special ***Anna debugger*** (also part of the Annalyzer tool suite). For the purpose of this paper, we are interested mainly in the capabilities of the Anna debugger. Details of how Anna formal comments are converted to runtime checking code are given in [16,17,18]. A complete user guide and installation manual is given in [14].

When a ***transformed*** Anna program is executed, the Anna debugger takes control and provides a top-level interface between the user and the program being tested. Control can be transferred to the underlying program in which case, control returns to the debugger when the program becomes inconsistent with some annotation. In addition to transferring control to the Anna debugger, the exception **ANNA-ERROR** is also raised. The debugger provides the following capabilities:

- ***Diagnostics.***
  Provides diagnostic messages when the program becomes inconsistent with an annotation. In this case, the annotation violated and the location of violation is displayed to the programmer.

- ***Manipulation*** *of* ***annotations.***
  Annotations can be suppressed or unsuppressed, and their effect when they are violated can be changed. For example, annotations can be completely suppressed, i.e. the program will behave as if the annotation were not present. This feature will be used in the example of Section 5.

The programmer interacts with the debugger using menus, choosing displayed options with an input device such as a mouse. An example of how the programmer may suppress annotations is shown in Figure 1. The first level menu displays the various different tools available in the tool suite. On choosing the Anna Debugger, a second level menu displays the various options the programmer has in interacting with the debugger. On deciding to suppress annotations, another menu displays the set of annotations that can be suppressed.

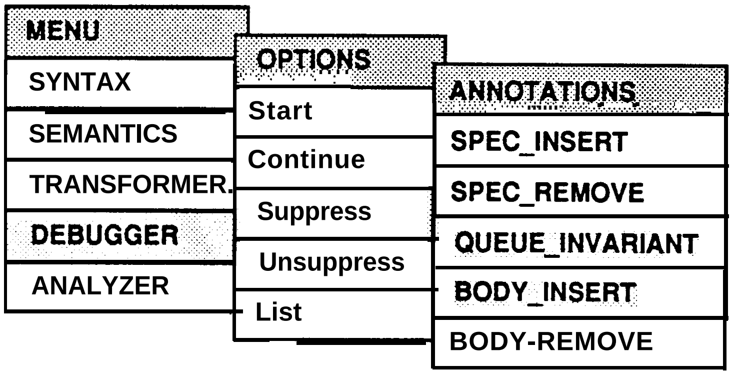
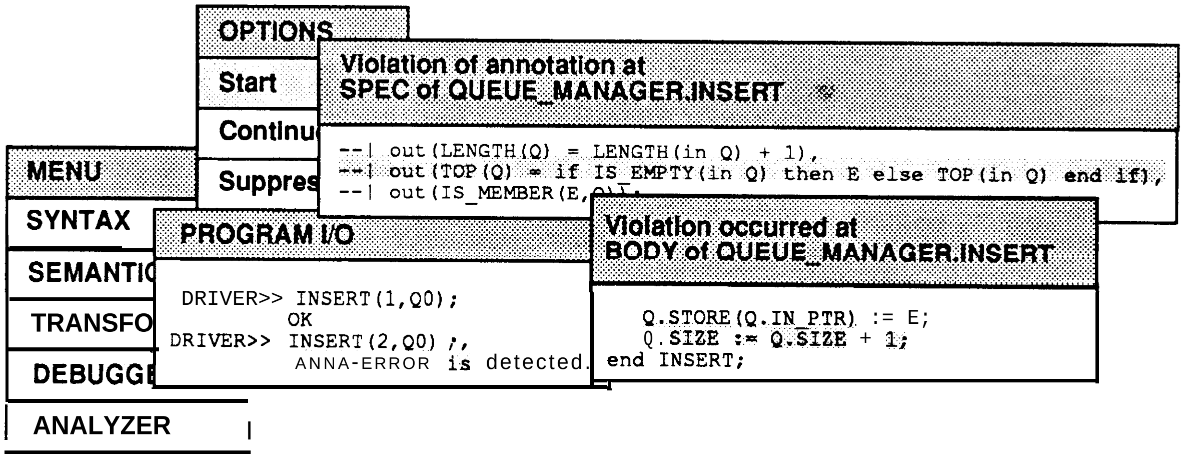


Figure 1: The Annalyzer Tool-Suite Menus.

The programmer can now choose the set of annotations to be suppressed and then commit this action.

In addition to the menus of Figure 1, there is a window that displays the program execution. When an annotation is violated, two more windows are opened. One of these windows shows the annotation violated while the other shows the local program text around the statement where it was violated. This scenario is illustrated in Figure 2. This figure corresponds to one of the scenarios of the example in Section 5.



```
--| out(LENGTH(Q) = LENGTH(in Q) + 1),
--| out(TOP(Q) = if IS_EMPTY(in Q) then E else TOP(in Q) end if),
--| out(IS_MEMBER(E,Q)),
```

```
 DRIVER>> INSERT(1,Q0);
          OK
DRIVER>> INSERT(2,Q0) ;,
          ANNA-ERROR is detected.
```

```
   Q.STORE(Q.IN_PTR) := E;
   Q.SIZE := Q.SIZE + 1;
end INSERT;
```

Figure 2: Error Reporting By The Anna Debugger.

4 Two Dimensional Pinpointing

Two dimensional pinpointing is a process for locating errors in programs. It starts **if** and when inconsistencies arise while the program is being tested. The pinpointing process attempts to reduce the region of suspicion based on the diagnostic messages reported by the *Annalyzer* and our understanding of the software being debugged. The ***region of suspicion*** is the portion of the program that is known to contain an inconsistency.

In this paper, we concentrate on the application of two dimensional pinpointing to debugging Ada packages. We assume that the package is complex with several structural levels. While the package executes, it performs a sequence of package operations. Details of the structural levels of packages are given in §4.1.

To debug a package, we first choose a test sequence — a sequence of package operations. The package is executed on these operations one after the other. If an inconsistency is detected, we start the process of two-dimensional pinpointing. At this point, only the specifications at the highest structural level are activated. The lowest structural level at which specifications are activated is the ***current level*** at which the pinpointing process is taking place.

As the term — two-dimensional pinpointing — suggests, the pinpointing process involves two "dimensions". These two dimensions are:

1. ***The length** of **the test sequence.***
2. ***The number of structural levels of abstraction.***

The pinpointing process is carried out by performing the following steps and re-executing the same test sequence repeatedly until the region of suspicion is reduced to a stage when repair of this region can be attempted. These steps are discussed further in §4.3.

1. ***Add new specifications at the current level** of **the pinpointing process.***
2. ***Activate specifications within the region** of **suspicion that are at the next lower structural level.***

After either of the two steps above, more annotations are added to the set being checked. Hence when the test sequence is re-executed, it is compared against a new set of annotations. A violation against these new annotations will give more information about the location of the problem. This helps reduce the region of suspicion. This will be illustrated in detail in Section 5.

Breadth-first search strategy: ***Always try to add specifications at the current level being tested before activating specifications at the next lower level.***
This leads to completion of the most global and important specifications, and reduces the region of suspicion quickly. Fewer annotations need be activated at the next lower level, if the region is reduced. The breadth-first paradigm allows us to abstract away from the low-level details of the implementation and consider only the relationships between high-level abstract entities.

4.1 Packages and Levels

A package can be considered to be a composition of entities at different ***structural levels.*** A simple package may have the following structural levels:

- ***The visible level:***
  The visible level of the package is where the interface of the package to the rest of the program is defined. The visible level typically consists of ***private type*** and ***subprogram*** declarations.

- ***The data level:***
  At this level is the definition of the data structures and the data objects of the package. The ***private part*** of the package is also part of the data level.

- ***The subprogram level:***
  This level consists of the actual bodies of the subprograms inside the package body.

- ***The statement level:***
  This level consists of the sequence of statements within the subprogram. Actually, this level can be split up into more than one level. For example, compound statements (e.g. loops) may be considered at a ***higher*** structural level than the simple statements within it.

In the example of Section 5, we have a Queue package with the standard operations of ***Insert, Remove,*** etc. The structural levels of this package is illustrated in Figure 3.

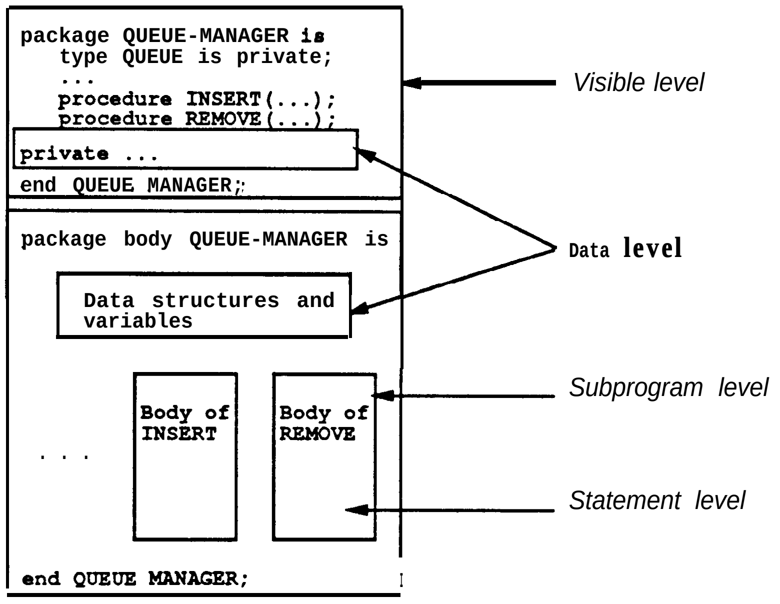


Figure 3: Structural Levels within the Queue Package.

The test sequences consist of visible package operations. In Section 5, the test sequences consists of repeated applications of the operations ***Insert*** and ***Remove.***

4.2 Assumptions about Specifications

The following assumptions about specifications (at the debugging stage) help us to focus the pinpointing methodology:

1. ***No initial (starting) specification may be changed.***
   We assume that the specifications have been written and analyzed at some earlier stage in the software development process. Hence for the purposes of the pinpointing methodology, we can assume that the initial specifications are correct and hence may not be changed.

2. ***Concepts used in the specifications are correct.***
   Concepts are functions used in specifications and annotations. We assume that bodies implementing all concepts have been debugged already and that they are correct.

   The following example illustrates the above assumptions:

   ```
       procedure INSERT( E : ELEMENT; Q : in out QUEUE);
   --| where
   --        out(LENGTH(Q) = LENGTH(in Q) + 1),
   --        out (IS_MEMBER( E, Q));
   ```

   In the above example, LENGTH and IS-MEMBER are concepts that have been used to specify INSERT[10]. While debugging, we assume that the specification of **INSERT** is correct and that the concepts **LENGTH** and **IS-MEMBER** are correctly implemented.

3. ***New specifications may be added provided they are consistent with the old ones.***
   Specifying complex software is a difficult task. It is quite possible that some undesired behavior has been overlooked and specifications against it have been omitted. We refer to such specifications ***as incomplete.*** Therefore, new specifications may be added to specify against undesired behavior that has been overlooked. We refer to this process ***as augmenting*** specifications.

4. ***Repair may involve any change to executable code, but data structures may be changed only to the extent that they are not constrained by the initial specifications.***
   This follows from our assumption made above that the initial specifications are correct.

4.3 Guidelines for Applying the Pinpointing Methodology on Ada Packages

We now present a set of guidelines to perform two dimensional pinpointing on Ada packages. These guidelines are illustrated in Section 5. A detailed treatment of the pinpointing process is given in [12].

The following guidelines are described with respect to the visible level of the package for the sake of clarity. Many of these guidelines apply equally well to other levels of the package also.

Visible package specifications may not be complete (this is one of the assumptions in our methodology). Due to this, we need to keep two situations in mind:

[10]So also is the Ada "+" on the INTEGER type.

- There might occur an inconsistency of the program behavior with an intuitive intention which hasn't been formally expressed.
- In a sequence test of a package, we cannot assume that the subprogram in which a violation occurs is at fault because the previous operations might influence (either directly or as a side-effect) the variables in the violated specification.

Our goal is to reduce the region of suspicion, either by reducing the length of the test sequence leading to a violation (first dimension), or by finding a lower level annotation that is violated (second dimension). There are three choices:

1. We try to add new annotations at the level being tested (here, the visible level). This choice is indicated whenever when we think that we can find missing annotations. It is obviously necessary whenever the program behavior is inconsistent with an intuitive intention although no annotation was violated. This strategy usually results in reducing the region of suspicion by shortening the test sequence in which a violation occurs.
2. We decide to test the package at the next lower level — in this case the data level. We activate the specifications at the data level. The region of suspicion will be reduced if a lower level annotation is violated. We take this choice when we cannot think of any missing visible level specification.
3. We decide that we have pinpointed the region of suspicion to the single subprogram which is violated. This case occurs in the following situations:
   (a) We cannot think of any further missing specification at the data level. Since package operations interact only through the data level, a complete set of specifications at the visible and data levels means that every operation is executed in a consistent package state. Hence a violation can only be due to a problem within the same operation.
   (b) We have confidence in all other operations in the test sequence except the violated operation (this can happen even at the visible level).
   (c) Subprogram annotations at the subprogram body level are violated before data level or visible level annotations.
   (d) The first subprogram in the test sequence is violated.

The following guidelines are more specific to the subprogram level. If we decide to take the third choice in the guidelines at the visible level, we can consider two cases:

- We try to pinpoint the region of suspicion at the statement level.
- We repair the body if we think that the region of suspicion is sufficiently pinpointed.

5 An Illustrative Debugging Session

In this section, the **QUEUE-MANAGER** package written in Ada/Anna is tested and debugged, using the Annalyzer. The example, although simple for brevity, illustrates the two-dimensional pinpointing methodology described in Section 4.3.

5.1 Description of the QUEUE-MANAGER Package

The **QUEUE-MANAGER** package provides an abstract queue type, procedures, functions, and exceptions. The Ada declaration of the package contains the abstract Anna specification which is visible to users. This Ada/Anna package visible specification defines the behavior of the facilities (types and operations) provided by the package.

The Ada package body, including the Ada private part, contains an implementation, the details of which are hidden from users. This hidden part, which has three structural levels, also contains local annotations specifying how the implementation works. The hidden annotations refer to the hidden implementation details, and thus will be different for different implementations.

The development of the **QUEUE-MANAGER** visible specification, and an implementation for it, goes roughly as follows:

1. Types and exceptions are declared. In particular, the abstract data type **QUEUE** is declared as an Ada private type.

2. The functions **IS-MEMBER, LENGTH,** and TOP are declared. We refer to them as basic concepts because they are used to specify all of the other **QUEUE-MANAGER** operations.

3. The rest of the operations are declared and specified in terms of the concepts. Actually, two of the other package operations, **IS-EMPTY** and **IS-FULL,** which have very simple specifications in terms of the basic concepts, are also used as concepts in specifications. This completes construction of the visible specification.

4. The abstract data type **QUEUE** is represented as a record type structure in the private part of the **QUEUE-MANAGER** package. This is the first and most critical decision in the implement ation. The values of this structure are subject to a type constraint which expresses a major decision about its use in this particular implementation.

5. The bodies of the concepts are implemented.

6. The rest of operations are implemented.

Here is the description of the actual package:

Specification of the QUEUE-MANAGER Package:

```
generic
   type ELEMENT is private;
   MAX : POSITIVE;
package QUEUE-MANAGER is
   type QUEUE is private;
```

```
        EMPTY, FULL : exception;

-- The following are concepts used in specifications. IS-EMPTY and IS_FULL are defined in
-- terms of LENGTH.

--:     function IS_MEMBER(E : ELEMENT; Q : QUEUE) return BOOLEAN;

        function LENGTH(Q : QUEUE) return INTEGER;

        function TOP(Q : QUEUE) return ELEMENT;

        function IS_EMPTY(Q : QUEUE) return BOOLEAN;
--|     <<SPEC_IS_EMPTY>> where
--|         return LENGTH(Q) = 0;

        function IS_FULL(Q : QUEUE) return BOOLEAN;
--|     <<SPEC_IS_FULL>> where
--|         return LENGTH(Q) = MAX;

-- The following are operations specified by concepts.

        procedure INSERT( E : ELEMENT; Q : in out QUEUE);
--|     < <SPEC_INSERT> > where
--|         IS-FULL(Q) => raise FULL,
--|         raise FULL => Q = in Q,
--|         out(LENGTH(Q) = LENGTH(in Q) + 1),
--|         out (IS_MEMBER( E, Q));

        procedure REMOVE(E : out ELEMENT; Q : in out QUEUE);
--|     < <SPEC_REMOVE> > where
--|         IS-EMPTY(Q) => raise EMPTY,
--|         raise EMPTY => Q = in Q,
--|         out(LENGTH(Q) = LENGTH(in Q) - 1),
--|         out(E = TOP(in Q));

--  axiom

        ...  -- Axiomatic specifications are omitted in this exam-
             -- ple. Debugging using algebraic axioms is described
             -- in [12].
```

```
   private
     type QUEUE-ARRAY is array(INTEGER range <>) of ELEMENT;
     type QUEUE is
        record
           STORE : QUEUE-ARRAY( 1. . MAX);
           IN-PTR, OUT-PTR: INTEGER range 1. . MAX := 1;
           SIZE :INTEGER range 0. . MAX := 0;
        end record;
--|  < <QUEUE-INVARIANT> > where
--|     in out Q:QUEUE =>
--|        (Q . IN-PTR – Q . OUT-PTR – Q . SIZE) mod MAX = 0;

   end QUEUE-MANAGER;
```

Implementation **of the QUEUE_MANAGER Package:**

```
   package body QUEUE-MANAGER;

     function IS_MEMBER(E : ELEMENT; Q : QUEUE) return BOOLEAN is
        I : INTEGER := Q. OUT-PTR;
     begin
        if IS_EMPTY(Q) then
          return FALSE;
        end if;
        loop
          if Q .STORE(I) = E then
            return TRUE;
          end if;
          I := I mod MAX+1;
          if I = Q .IN_PTR then
            return FALSE;
          end if;
        end loop;
     end IS-MEMBER;

     function LENGTH(Q : QUEUE) return INTEGER is
  |  < <BODY-LENGTH> > where
  |     return Q. SIZE;
     begin
        return Q. SIZE;
     end LENGTH;
     function TOP(Q : QUEUE) return ELEMENT is
--|  <<BODY-TOP>> where
--|     return Q . STORE(Q . OUT-PTR);
     begin
        if IS-EMPTY(Q) then
           raise EMPTY;
        else
           return(Q . STORE(Q . OUT-PTR));
        end if;
     end TOP;

     function IS_EMPTY(Q : QUEUE) return BOOLEAN is
--|  < <BODY-IS-EMPTY > > where
--|     return Q .SIZE = 0;
     begin
        return Q.SIZE = 0;
     end IS-EMPTY;

     function IS_FULL(Q : QUEUE) return BOOLEAN is
--|  < <BODY-IS-FULL> > where
--|     return Q .SIZE = MAX;
     begin
        return Q.SIZE = MAX;
     end IS-FULL;

     procedure INSERT( E : ELEMENT; Q : in out QUEUE) is
     begin
        if IS-FULL(Q) then
           raise FULL;
        end if;
        Q . STORE(Q. IN-PTR) := E;
        Q.SIZE := Q.SIZE + 1;
     end INSERT;

     procedure REMOVE( E : out ELEMENT; Q : in out QUEUE) is
     begin
        if IS-EMPTY(Q) then
           raise EMPTY;
        end if;
```

```
    E := Q . STORE(Q. IN-PTR);
    Q. OUT-PTR := Q.OUT_PTR mod MAX+1;
    Q . SIZE := Q.SIZE - 1;
  end REMOVE;

end QUEUE-MANAGER;
```

5.2 A Debugging Session

Now we illustrate a session with the Annalyzer aimed at testing and debugging this particular QUEUE-MANAGER specification and body. The session consists of six ***interactions*** in which tests are made, results are analyzed, and further actions are taken. Each interaction demonstrates how formal specifications are used in the two-dimensional pinpointing methodology. The interactions are now shown below:

Interaction 1

Test

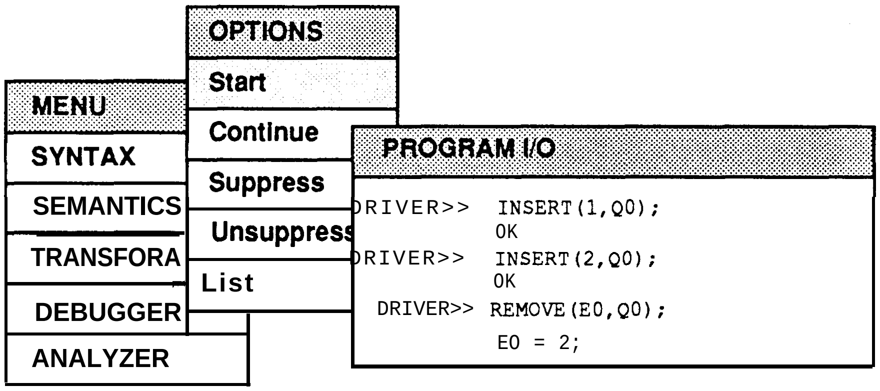
We declare a queue variable **QO** and an element variable EO. We also suppress all private part annotations and package body annotations to test the behavior of the package body for consistency with visible specifications.

We execute a test sequence of three calls to package operations:

INSERT (1, QO); INSERT (2, QO); REMOVE(EO, QO);

Result

No Anna violation occurred at the visible level. But an inconsistency with our intuitive intention exists since 2 was removed from QO, whereas the first element inserted was 1. See the Program I/O window in Figure 4.



```
DRIVER>>  INSERT(1,Q0);
          OK
DRIVER>>  INSERT(2,Q0);
          OK
DRIVER>>  REMOVE(E0,Q0);
          E0 = 2;
```

Figure 4: Result 1.

Explanation: An incomplete visible specification.

This happened because the visible specification does not express all of our intuitive requirements of queues — clearly something has been forgotten. At this stage, an unexpected result has occurred at the end of the sequence,

```
    INSERT(1,Q0);INSERT(2,Q0);REMOVE(E0,Q0);
```

when the interaction level is the package visible level. The region of suspicion is the visible specifications of **INSERT** and **REMOVE,** the data level, and the bodies of these package operations. See Figure 10 in page 29. The reason to include **INSERT** in the region of suspicion is given in the general guideline stated in Section **4.3.**

Guideline 1

Pinpointing the region of suspicion at the visible level

Starting with the operation that was violated, and working back through the test sequence:

1. ***Express the violated intuitive requirement as a new formal annotation.***
2. ***Consider how the previous operations of the test influence the violated requirement. If we can think of any missing specification of the previous operations that is related to the violated one, express it as a formal annotation. Missing specifications at the debugging stage are often invariants of visible operations.***

Action & Justification

We decide to take the first choice in the general guideline stated in Section 4.3. We have to consider why this inconsistency occurred and look for missing visible specifications. First we guess

- REMOVE didn't return the value TOP(Q0).

However, we see that the visible specification of REMOVE contains

```
 --       out (E = TOP(in Q));
```

and that it wasn't violated. Hence, our first guess was wrong since we assume the correctness of TOP.

Next, we guess

- The value TOP(Q0) was changed in one of the executions of INSERT.

Now we consider what the expected behavior of INSERT with respect to the concept TOP is. We realize that TOP(Q) for a non-empty Q must remain invariant under an INSERT operation, and that this is missing in the visible specification of INSERT. This missing specification is formally expressed as

```
    procedure INSERT(E : ELEMENT; Q : in out QUEUE);
--|<<SPEC_INSERT>>
--|where
--|    out(TOP(Q) = if IS_EMPTY(inQ) then E else TOP(inQ) end if);
```

We add this to the other annotations of **INSERT** and run the same test sequence, in order to check whether our second guess is correct or not.

Remarks

1. When we apply two-dimensional pinpointing methodology at the visible level, it is important not to reason about the cause of an inconsistency based on a particular implementation detail such as elements being placed in some positional order in a common data structure as an array or a linked list. At the visible level of QUEUE-MANAGER, all available to us are the operations LENGTH, TOP, IS-MEMBER, IS-EMPTY, IS-FULL, INSERT, and REMOVE, and the abstract relations among these operations. For example, the reader should not think that the value of TOP(Q0) is the first element in an array.
2. Note that we could prove formally that the new annotation of INSERT in the visible package specification must be violated by the sequence test.
3. People often forget invariants like the above one.
4. Imagine what we have to do in order to check whether the above guesses are correct or not, using a standard debugger. We have to print out the values of low level variables in the package body, and reconstruct the values of the visible level concepts either in our mind or by using pencil and paper. Using the Annalyzer, it is automatically checked whether the guesses are correct or not, once we express the guesses as visible formal annotations of the appropriate subprograms.

Other Possible Actions

1. Another possibility considered was to add

```
        procedure INSERT(E : ELEMENT; Q : in out QUEUE);
    --|<<SPEC_INSERT>>
    -- |where
    --|   out (for all El :ELEMENT =>
    --|      (IS_MEMBER(E1,Q) <-> E1=E or IS_MEMBER(E1,in Q)));
```

   This was considered because we wondered whether INSERT deleted some elements or not. However this was rejected because it does not constrain against INSERT changing the value of **TOP,** i.e., this specification does not imply the specification which was added in the action we took.
2. Similarly, we might consider introducing a new specification concept, LAST, and specifying that **INSERT** must insert the new element into the last position of the queue[11]. However, it does not help.
3. We can unsuppress lower level annotations, and run the same test sequence; but this would violate our general breadth-first search strategy.

Interaction 2

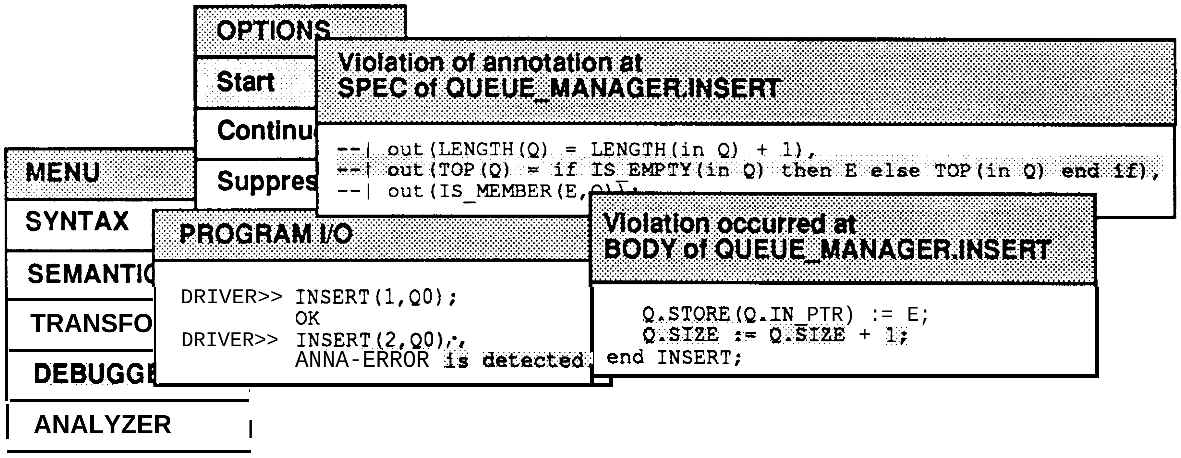
Test

We run the same sequence test again; but this time we only get as far as the second operation, INSERT(2, QO).

[11]This is a popular choice for many people.

Result

The Annalyzer detects a violation of our new specification and gives the following messages in the different windows. See Figure 5.



Violation of annotation at
SPEC of QUEUE_MANAGER.INSERT

```
--| out(LENGTH(Q) = LENGTH(in Q) + 1),
--| out(TOP(Q) = if IS_EMPTY(in Q) then E else TOP(in Q) end if),
--| out(IS_MEMBER(E,Q)),
```

```
DRIVER>> INSERT(1,Q0);
         OK
DRIVER>> INSERT(2,Q0);
         ANNA-ERROR is detected.
```

Violation occurred at
BODY of QUEUE_MANAGER.INSERT

```
   Q.STORE(Q.IN_PTR) := E;
   Q.SIZE := Q.SIZE + 1;
end INSERT;
```

Figure 5: Result 2.

Explanation

As we guessed, INSERT has an additional effect (beyond what is specified). The window displaying a violated annotation shows that the value of **TOP** is not invariant under INSERT's into non-empty queues. Note that it is the second INSERT operation that violates. See the Program I/O window in Figure 5.

The length of the sequence where a violation occurred has decreased. As a result, the region of suspicion is now reduced to the visible specification of **INSERT,** the data level, and the body of **INSERT.** See Figure 10 in page 29. This does not mean that REMOVE is correct, however. It means simply that an inconsistency between the annotations and the program text exists in this region.

Guideline 2

Pinpointing the region of suspicion at the data or body level.

Assume that we take the second choice stated in the general guideline in Section *4.3.* The following guidelines are useful:

1. ***Test the subprogram against the existing lower level annotations such as data invariants in private part or body, subprogram body annotations, etc.***
2. ***Add an annotation of the subprogram body which is a transformation of the visible specification that was violated. In transforming a visible annotation into an equivalent hidden one, abstract variables are replaced by their lower level (hidden) representations. Such an annotation can be used as a starting point for further pinpointing at the body level. It can be used in other situations, too. See the guideline in the Interaction 3.***
3. ***Add more detailed subprogram annotations to the subprogmm body. A good candidate is an annotation of the subprogram body referring to hidden com-***

ponents or variables; such detailed annotations are not expressible as subprogram annotations at the visible level.

Action & Justification

We decide to take the second choice in the guidelines stated in Section 4.3. Now our aim is to reduce the region of suspicion, interacting at the next lower level in the package structure, which is the data level. For this purpose, we unsuppress QUEUE-INVARIANT in the private part, and run the same test sequence (See the private part of QUEUE-MANAGER specification. Also, remember that previously the consistency checking of QUEUE-INVARIANT was **sup**pressed.) QUEUE-INVARIANT must be satisfied by all variables (and parameters) of a QUEUE record whenever a package operation terminates normally (see Section 2). Notice that at the (lower) data level the structure and components of queues are visible.

Remarks

1. With a standard debugger, we interact with the program at the same level as the low-level information shown in the window displaying the location of a violation. On the other hand, in our methodology, we interact with the program at the same level as the high-level specification shown in the window displaying a violated annotation. At this stage of the two-dimensional pinpointing process, we don't understand yet how the low-level information relates to the high-level specification.
2. The same remark as stated in Interaction 1 is applicable here. With a standard debugger, we have to print out the values of QO . IN-PTR, QO . OUT-PTR and QO . SIZE before and after each execution of the operation INSERT, in order to check whether QUEUE-INVARIANT is satisfied or not. On the other hand, with the Annalyzer, this invariant is checked automatically.

Interaction 3

Test

We run the same sequence test again.

Result

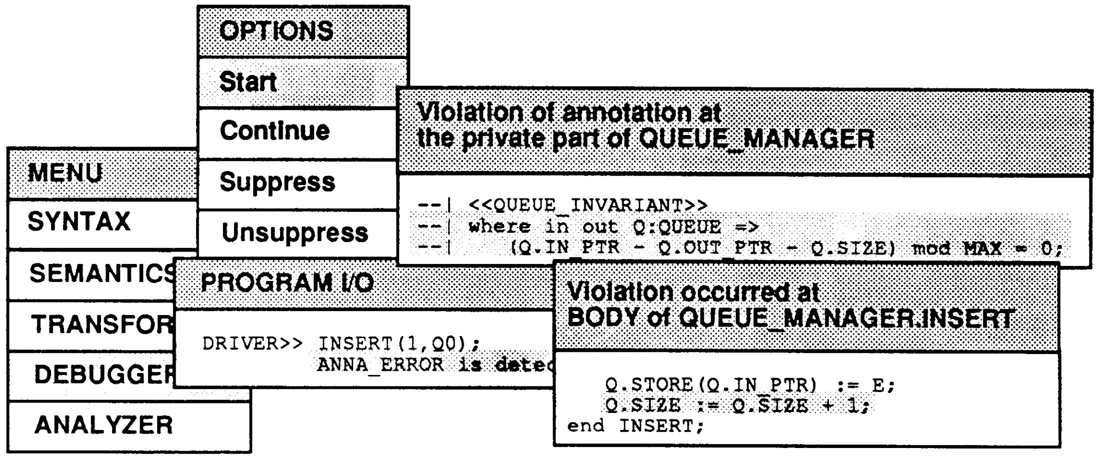
Anna consistency checking system detects a violation at the first call of INSERT and gives the following messages in the different windows. See Figure 6.

Explanation

This means that after an element was inserted, the invariant condition among Q . IN-PTR, Q . OUT-PTR, and Q . SIZE was violated. Notice that both dimensions, the ***length*** of the violated sequence and the ***level*** of the violated annotation are reduced. We have reduced the region of suspicion to the data level and the body of INSERT. See Figure 10 in page 29.

Guideline 3

Pinpointing the region of suspicion within a single subprogram or repairing a body.



Violation of annotation at
the private part of QUEUE_MANAGER

```
--| <<QUEUE_INVARIANT>>
--| where in out Q:QUEUE =>
--|     (Q.IN_PTR - Q.OUT_PTR - Q.SIZE) mod MAX = 0;
```

```
DRIVER>> INSERT(1,Q0);
         ANNA_ERROR is detec
```

Violation occurred at
BODY of QUEUE_MANAGER.INSERT

```
   Q.STORE(Q.IN_PTR) := E;
   Q.SIZE := Q.SIZE + 1;
end INSERT;
```

Figure 6: Result 3.

Once we decide to take the third choice in the guideline stated in Section 4.3, we have to reduce the region of suspicion in a single subprogram body or repair the body. In these cases, the following guidelines are useful:

1. *Add a subprogram annotation to the subprogram body which is a transformation of the visible specification that was violated*[12]. *This annotation can be used for further pinpointing as well as for repair.*
2. *Add more detailed subprogram annotations to the subprogram body*[13].
3. *Add statement annotations such as loop invariants or assertions within the code of the subprogram body.*
4. *At the subprogram body level, a considerable number of detailed annotations for a subprogram body are available. Thus, it is sometimes advisable to rewrite the body that is under suspicion, using these annotations, rather than to find the cause of the inconsistency in the body and to repair it.*

Action & Justification

We decide to repair the body of INSERT. At this stage we consider the goal plan for the body of INSERT. We consider how Q.IN_PTR, Q.OUT_PTR, and Q.SIZE have to be updated in the body of INSERT (See the body of INSERT in QUEUE_MANAGER body.). INSERT must achieve four goals:

1. **out**(Q.STORE(**in** Q.IN_PTR) = E)
   -- inserting E in STORE
2. **out**(Q.SIZE = **in** Q.SIZE + 1)
   -- incrementing LENGTH
3. **out**(Q.IN_PTR = **in** Q.IN_PTR **mod** MAX + 1)
   -- incrementing IN_PTR
4. <<QUEUE_INVARIANT>>

[12]See guideline 2, number 2.
[13]See guideline 2, number 3.

The orders of the first two and the last two are indeterminate. Since the fourth goal was violated, the body did not achieve it. We have to find a way of fixing the body so that the body achieves the fourth goal. Looking at the body, we realize that the body achieves the first and second goals, but not the third one. We can informally conclude that if we add the third goal as an assignment statement to the body, then the fourth goal is also met. Thus *Repair:* we add the following assignment statement to the body of INSERT.

```
Q . IN-PTR := Q.IN_PTR mod MAX + 1;
```

Other Possible Actions

1. We could pinpoint the inconsistency further in the body of INSERT by adding the above goals to the subprogram body as assertions at appropriate points.
2. The correctness of this repair also could be checked formally by proof methods. For example see [8].

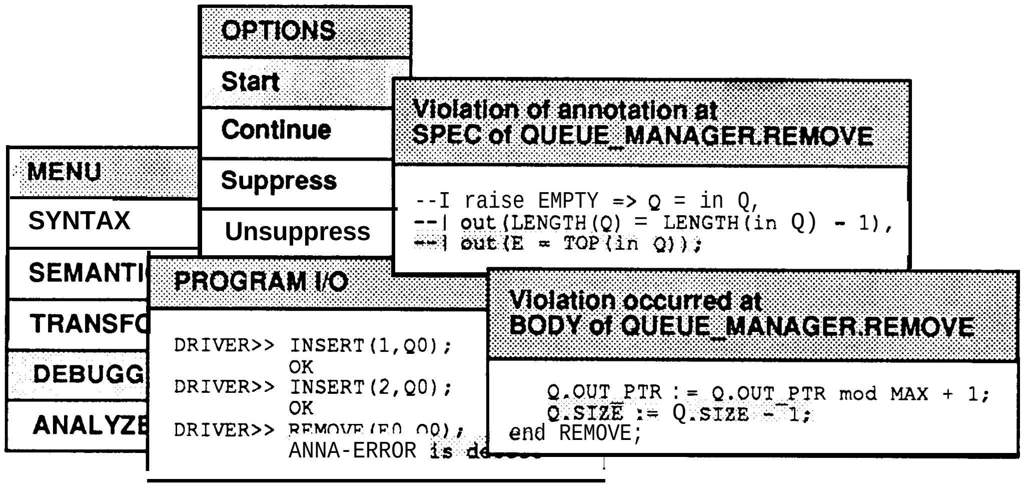
Interaction 4

Test

We run the same sequence test again, checking annotations at all levels.

Result

The Annalyzer detects another violation at the visible level and gives the following messages in the different windows. See Figure 7.



Violation of annotation at
SPEC of QUEUE_MANAGER.REMOVE

```
--I raise EMPTY => Q = in Q,
--| out(LENGTH(Q) = LENGTH(in Q) - 1),
--| out(E = TOP(in Q));
```

```
DRIVER>> INSERT(1,Q0);
         OK
DRIVER>> INSERT(2,Q0);
         OK
DRIVER>> REMOVE(E0,Q0);
         ANNA-ERROR is de
```

Violation occurred at
BODY of QUEUE_MANAGER.REMOVE

```
    Q.OUT_PTR := Q.OUT_PTR mod MAX + 1;
    Q.SIZE := Q.SIZE - 1;
end REMOVE;
```

Figure 7: Result 4.

Explanation

This test actually serves two purposes:

1. Running INSERT(1, QO); INSERT(2, QO) against the annotations at all levels in order to find out whether the repair we made in the previous interaction is correct on this test or not.

2. Running INSERT(1, QO); INSERT(2, QO); REMOVE(E0, QO) against visible specifications for further testing.

The result shows that INSERT, as repaired, is consistent with all existing specifications on this particular test sequence, but that another inconsistency occurred with a visible level specification. The region of suspicion is the visible specifications of INSERT and REMOVE, the data level, and the bodies of these procedures. See Figure 10 in page 29.

Action & Justification

The test sequence leading to a new violation and the level of the violated annotation is the same as in Interaction 1. However, we now know that INSERT is consistent on this test with all existing annotations at all levels. Also we make a judgement that the annotations of INSERT are complete, i.e., that we cannot think any further missing specifications. Hence, we have a good confidence in the correctness of INSERT. Therefore, following the third choice in Section 4.3, we decide that we have reduced the region of suspicion to the visible level of REMOVE, the data level, and the body of REMOVE. See Figure 10. Our aim is now to reduce the region of suspicion to the data level and the body of REMOVE. At this stage, if we strictly follow two-dimensional pinpointing, first we must consider finding an inconsistency at the data level. If we tried this, we would find no inconsistency. We omit the explanation of this interaction and consider further reducing the region of suspicion to the body of REMOVE. Now we must provide more detailed body annotations for REMOVE. We follow guideline 3, number 1. The violated visible annotation is

```
--|    out(E = TOP( in Q));                    (1)
```

This tells us exactly what the value of E in REMOVE should be. The body level annotation of TOP is

```
--     return Q . STORE(Q . OUT-PTR);          (2)
```

We transform the violated visible specification (1) into the body annotation of REMOVE, using (2). The resulting annotation is

```
        procedure REMOVE(E : out ELEMENT; Q : in out QUEUE)
    --| <<BODY-REMOVE>>
    --| where
    --|     out(E = in (Q. STORE(Q. OUT-PTR)));
```

This transformed annotation tells us exactly how E should be updated in the body of REMOVE. We add this transformed annotation to the body and run the same test.

Other Possible Action

We could check the actual out-put value of E on this test, using the Annalyzer options. The detail of such options will be explained in [14].

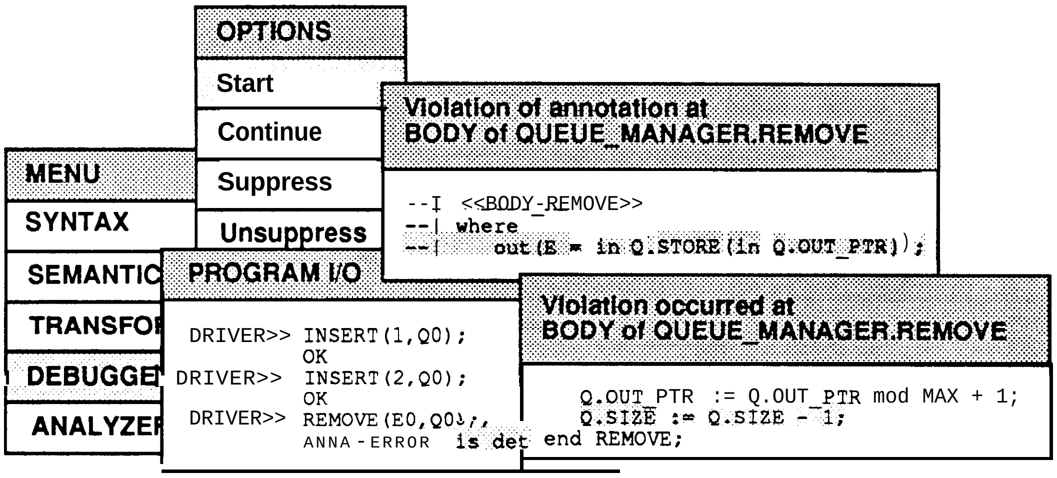
Interaction 5

Test

We run the same sequence test again.

Result

The Annalyzer detects a violation of the new body annotation of REMOVE, and gives the following messages in the different windows. See Figure 8.



Violation of annotation at
BODY of QUEUE_MANAGER.REMOVE

```
--| <<BODY-REMOVE>>
--| where
--|     out(E = in Q.STORE(in Q.OUT_PTR));
```

```
 DRIVER>> INSERT(1,Q0);
          OK
DRIVER>>  INSERT(2,Q0);
          OK
 DRIVER>> REMOVE(E0,Q0);,
          ANNA-ERROR is det
```

Violation occurred at
BODY of QUEUE_MANAGER.REMOVE

```
   Q.OUT_PTR := Q.OUT_PTR mod MAX + 1;
   Q.SIZE := Q.SIZE - 1;
end REMOVE;
```

Figure 8: Result 5.

Explanation

We have reduced the region of suspicion to the body of REMOVE . See Figure 10 in page 29.

Action & Justification

The annotation BODY-REMOVE tells us the exact component in Q . STORE to be returned as the value of E. We read the body of REMOVE and realize that the following repair is needed: *Repair*: *We* change IN-PTR to OUT-PTR in E := Q. STORE(Q. IN-PTR);.

Other Possible Action

The correctness of this repair could also be checked formally by proof methods.

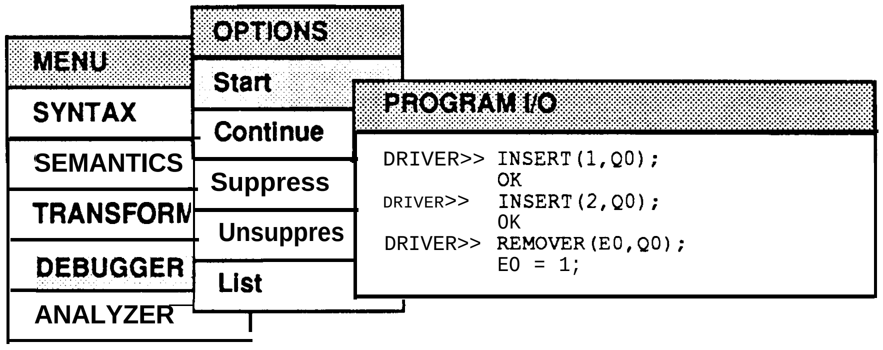
Interaction 6

Test

After fixing the bug as suggested in the previous step, we run the same test sequence.

Result

This time, no violation is detected. See Figure 9.



```
DRIVER>> INSERT(1,Q0);
         OK
DRIVER>> INSERT(2,Q0);
         OK
DRIVER>> REMOVER(E0,Q0);
         E0 = 1;
```

Figure 9: Result 6.

6 Conclusions

Two-dimensional pinpointing is a methodology for locating errors in Ada packages or similar complex software modules that are constructed using data abstraction and information hiding principles. It utilizes annotations at all levels of the software, starting at the most abstract level specification.

The methodology is independent of any compiler or runtime environment. It is supported by tools that transform annotations into runtime checks. Support tools have been developed for Ada and Anna, and can be clearly be developed for similar modern programming and specification languages.

We believe that this kind of use of abstract specifications in testing and debugging will become commonplace as programming/specification languages become more powerful, and analysis tools such as the Annalyzer become readily available. It is particularly appropriate in reuse of software, where for example, a standard interface specification may be implemented many times to fit differing environments, or modified by additional requirements.

Work towards developing pinpointing methods is continuing in several directions. First, pinpointing is being applied to more complex packages than the simple example shown here. This involves a theory of building correct specifications, and also the checking of algebraic specifications [12]. One objective is to develop detailed guidelines covering many situations as a step towards constructing rule-based automated aids to debugging.

Secondly, use of formal proof in debugging and repair is being considered. Although the presentation here utilizes intuitive judgements and guesswork, formal proof methods can be employed to justify various steps in two-dimensional pinpointing.

Finally, development of similar pinpointing methods for concurrent software is underway.

7 Acknowledgements

Substantial portions of the Anna toolsuite have been implemented by Geoff Mendal. Other basic Anna environment tools used in this work were implemented by David Rosenblum, Chuan-Chieh Ko, and other members of the Program Analysis and Verification Group.

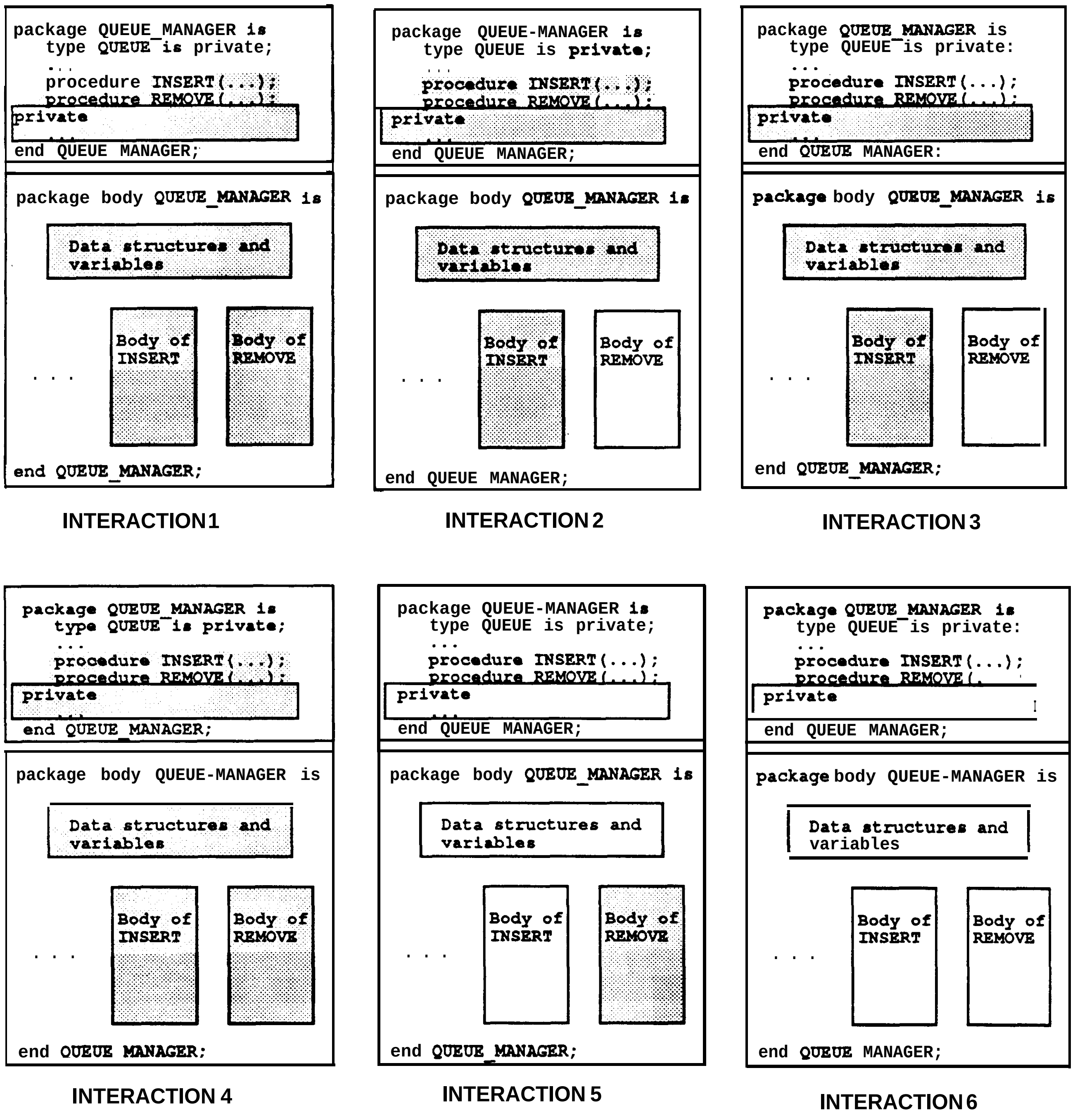


Figure 10: Region of Suspicion at each Interaction

References

[1] The *Ada Programming Language Reference Manual.* US Department of Defense, US Government Printing Office, February 1983. ANSI/MIL-STD-1815A-1983.

[2] F. L. Bauer, B. Moller, M. Partsch, and P. Pepper. Formal program construction by transformations-computer-aided, intuition-guided programming. *IEEE Transactions on Software Engineering,* 15(2):165–180, February 1989.

[3] D. Bjorner and C. Jones. *Formal Specification and Software Development.* Prentice Hall, 1982.

[4] D.C. Luckham. Program verification and verification-oriented programming. In *Proceedings of IFIP Congress* 77, pages 783-793, North-Holland Publishing Co., Amsterdam, August 1977.

[5] D.C. Luckham, D.P. Helmbold, S. Meldal, D.L. Bryan, and M.A. Haberler. *TSL: Task Sequencing Language for Specifying Distributed Ada Systems: TSL-1.* Technical Report CSL-TR-87-334, Stanford University, July 1987. Program Analysis and Verification Group Report PAVG-34.

[6] D.C. Luckham and F.W. von Henke. An overview of Anna, a specification language for Ada. *IEEE Software,* 2(2):9–23, March 1985.

[7] Richard A. DeMillo, W. Michael McCracken, R. J. Martin, and John F. Passafiume. *Software Testing and Evaluation.* Benjamin/Cummings, 1987.

[8] David Gries. *The Science of Programming. Texts and Monographs in Computer Science*, Springer-Verlag, 1981.

[9] John V. Guttag, James J. Horning, and Jeannette M. Wing. The Larch family of specification languages. *IEEE Software,* 2(5):24–36, September 1985.

[10] William C. Hetzel, editor. *Program Test Methods. Series in Automatic Computation*, Prentice-Hall, 1973.

[11] David C. Luckham, Randall B. Neff, and David S. Rosenblum. *An Environment for Ada Software Development Based on Formal Specification.* Technical Report 86-305, Computer Systems Laboratory, Stanford University, August 1986. (Program Analysis and Verification Group Report 31).

[12] David C. Luckham, Sriram Sankar, and Shuzo Takahashi. The methodology of formal specification and hierarchical debugging. (In preparation).

[13] David C. Luckham, Friedrich W. von Henke, Bernd Krieg-Brückner, and Olaf Owe. *Anna-A Language for Annotating Ada Programs.* Springer-Verlag-Lecture Notes in Computer Science No. 260, July 1987. (Also Stanford University Computer Systems Laboratory Technical Report No. 84-261).

[14] Geoff Mendal et al. The Anna-l user guide and installation manual. Computer Systems Laboratory, Stanford University, Stanford, California - 94305. Jan. 1989. Available upon request. Forthcoming Technical Report.

[15] M. Nielsen, K. Havelund, K. R. Wagner, and C. George. The RAISE language, method and tools. In *Proceedings of the VDM Conference,* pages 376-405, Springer-Verlag— Lecture Notes in Computer Science No. 328, 1988.

[16] Sriram Sankar. Automatic runtime consistency checking and debugging of formally specified programs. (forthcoming PhD thesis).

[17] Sriram Sankar and David S. Rosenblum. *The Complete Transformation Methodology for Sequential Runtime Checking of an Anna Subset.* Technical Report 86-301, Computer Systems Laboratory, Stanford University, June 1986. (Program Analysis and Verification Group Report 30).

[18] Sriram Sankar, David S. Rosenblum, and Randall B. Neff. An implementation of Anna. In *Ada in Use: Proceedings of the Ada International Conference, Paris,* pages 285-296, Cambridge University Press, May 1985.

[19] D. Scott and W. Scherlis. First steps towards inferential programming. In *Proceedings of the IFIP Conference,* pages 199-212, 1983.

[20] Raymond T. Yeh, editor. *Current Trends in Programming Methodology, Volume 2— Program Validation.* Prentice-Hall, Inc., 1977.