Approaches to Specification-Based Testing

Debra J. Richardson
Owen O'Malley
Cindy Tittle

Information and Computer Science
University of California
Irvine, CA 92717

Abstract

Current software testing practices focus, almost exclusively, on the implementation, despite widely acknowledged benefits of testing based on software specifications. We propose approaches to specification-based testing by extending a wide variety of implementation-based testing techniques to be applicable to formal specification languages. We demonstrate these approaches for the Anna and Larch specification languages.

1 Introduction

Specifications provide valuable information for testing. Most software testing techniques, however, rely solely on the implementation for information upon which to select test data. These implementation-based testing techniques focus on the actual behavior of the implementation but ignore intended behavior, except inasmuch as test output is manually compared against it. On the other hand, considering information from formal specifications enables testing intended behavior as well as actual functionality. Specification-based testing techniques may direct attention to aspects of the problem that have been implemented incorrectly or completely neglected, while implementation-based techniques reveal such aspects only by chance.

Specification-based testing techniques should be used to augment, not replace, implementation-based, or structural, testing. If only the specification is considered, then implementation-specific details, such as data structures and algorithms, are not sufficiently tested. We propose explicit interaction between specification-based and implementation-based testing techniques.

It is widely recognized that the software lifecycle must include validation activities in each phase, including quality assessment of the developed documents and test case selection based on these documents. Figure 1 shows a software lifecycle model where quality assessment and test case selection are pervasive[1]. At each phase, the selected test cases provide a test plan focusing on the level of abstraction considered in that phase. As soon as executable software is developed, appropriate test cases are run.

We focus here on utilizing specifications in selecting test cases. Specification-based test case selection has traditionally consisted of user-selected test cases based on a requirements specification. When the specification is informal, as is too often the case, this is effectively all that can be done. Existing specification-based testing systems manage user-selected test cases [OSW86], but automated test case selection is not feasible. In the later stages of the lifecycle, formal specification languages can be used and formal specification-based testing techniques could be employed. In particular, integration and unit test cases can be selected from formal module specifications. Unfortunately, very few such techniques exist. It is on the shaded boxes in Figure 1, specification-based and design-based test case selection, that we focus in the research described here.

In this paper, we describe our research aimed at developing approaches to specification-based test case selection. Functional[2] testing based on requirements has been done since the dawn of programming, but has consisted merely of heuristic criteria [Mye79]. It is difficult to determine when and if such criteria are satisfied. This was in part the motivation for developing implementation-based testing techniques; they have the advantage that their application can be automated and their satisfaction determined. Many of the implementation-based techniques are formalizations of the functional testing heuristics — for instance, domain testing [WC80] is a formal variant of boundary value testing [Mye79]. We propose to complete the circle and formalize specification-based testing.

We believe that traditional functional testing can be formalized by extending implementation-based techniques to be applicable with formal specification languages. The approaches we are developing are applicable to a spectrum

This work was supported in part by the National Science Foundation under grant CCR-8704311, with cooperation from the Defense Advanced Research Projects Agency (ARPA Order 6108, Program Code 7T10).

[1]Note that this is not a complete model of the lifecycle; it does not, for instance, include other analysis activities or maintenance.

[2]Functional testing has been given two distinct interpretations over the years. First, it referred to testing guidelines based on requirements [Mye79]. Howden gave it another meaning in his functional testing [How86].

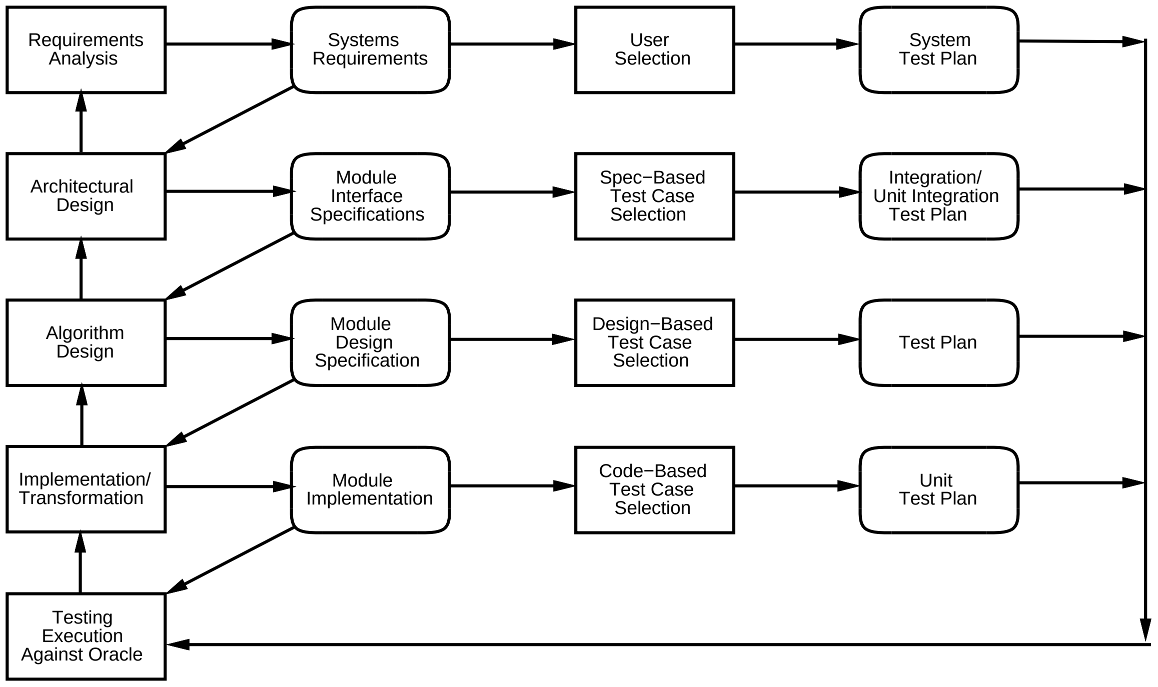
Requirements Analysis
Systems Requirements
User Selection
System Test Plan
Architectural Design
Module Interface Specifications
Spec−Based Test Case Selection
Integration/ Unit Integration Test Plan
Algorithm Design
Module Design Specification
Design−Based Test Case Selection
Test Plan
Implementation/ Transformation
Module Implementation
Code−Based Test Case Selection
Unit Test Plan
Testing Execution Against Oracle

Figure 1: Test Case Selection Throughout The Software Lifecycle

of languages, including algebraic specifications, state-based specifications, pre/post conditions, assertions, and procedural design languages. We are also considering a wide variety of functional and implementation-based techniques — not simply coverage techniques.

In the next section, we describe the predominant testing methods, categorizing them by the types of errors they uncover. In section three, we briefly overview past research in specification-based testing. In section four, we describe four approaches to specification-based testing developed by extending the notions underlying error-based and fault-based testing techniques, which have typically been based on implementation source code. The first two approaches postulate that the specification may be incorrect or that it may influence development of an incorrect implementation. The second two "assume" that the specification is a correct oracle. Sections five and six apply the approaches to two specification languages: Anna and Larch. In conclusion, we discuss our intentions for future research.

2 Implementation Based Testing

Most software testing work considers structural, or white box, unit testing — that is, independent testing of a single procedure based on information garnered from analyzing the procedure's implementation. Many structural techniques simply address path selection, while others are concerned with test data selection as well.

Path selection techniques are concerned with which statements or combinations of statements should be executed for increased error detection. The most common path selection techniques are *control flow coverage* techniques. *Data flow coverage* is proposed as more sensitive to error detection. A survey and graph-theoretic analysis of several path selection techniques appears in [CPRZ89]. In general, a path selection technique must be augmented by a test data selection technique, many of which select data for a specific path. Without judicious selection, test data may inadvertently mask faults in the source code — a phenomenon called *coincidental correctness*. We classify test data selection techniques as *error-based* or *fault-based*.

Error-based techniques are geared toward revealing specific types of errors, where an error is an incorrect value[3] produced by program execution. Formal error-based techniques analyze a (partial) path representation, usually developed by symbolic evaluation [RC85b], and select test data expected to be sensitive to specific types of errors. A survey of error-based testing appears elsewhere [CR83]. *Domain testing* focuses on the detection of domain errors by analyzing the path condition and selecting data on the boundaries of the path domain [WC80]. *Computation testing* analyzes the path computation and selects special values for the algebraic manipulations [How78, CR83].

Fault-based testing selects test data that focus on detecting particular types of faults, where a fault is a mistake in the source code. Fault-based testing techniques consist of "rules" that are applied to the source code to select test data sensitive to commonly-introduced faults. A survey of fault-based testing appears elsewhere [RT86]. The earliest formalized fault-based testing techniques were introduced independently by Hamlet [Ham77] and by DeMillo, Lipton and Sayward [DLS78]. These techniques, and those that followed, attempt to distinguish the program being tested from *variants* in a set of related programs that differ by the defined types of faults. The RELAY model[4] provides a fault-based

[3]An error may be a wrong internal value; an observable error is a failure.

[4]Relay is so named because of its analogy with a relay race.

technique for test data selection [RT88]. RELAY guarantees the detection of errors caused by any fault in a user-chosen fault classification. The RELAY model proposes the selection of test data that *originates* an error (introduces an incorrect state) for a potential fault of some type and *transfers* that error along some *route* through computations and data flow until a failure is *revealed.* RELAY develops *revealing conditions* that describe how to distinguish the source from the variant. Any test data set satisfying the revealing conditions contains some test datum that reveals the chosen faults.

We are extending error-based and fault-based techniques to be applicable to formal specification languages. Two approaches, spec/error-based testing and spec/fault-based testing, consider that the specification may be faulty or may be the source of faults introduced into the implementation and apply error-based or fault-based techniques to the specification. The other two approaches, oracle/error-based testing and oracle/fault-based testing, treat the specification as an oracle to be violated, while error-based or fault-based techniques are applied to the implementation in the hopes of detecting errors. We describe each of these approaches below and then provide several examples of their application to Anna and Larch in the sections that follow.

3 Previous Work in Specification-Based Testing

It has long been acknowledged that test case selection should be based on more than merely the source code [GG75]. Gourlay provides a mathematical framework for testing that confirms the need for specification-based testing [Gou83]. To achieve well-understood results from specification-based testing, however, the specifications must be written in a formal language with well-defined semantics. Laski illustrates that informal specifications fail to uncover errors [Las88].

The traditional functional testing approach is to partition the input domain into equivalence classes and select test data from each class. Goodenough and Gerhart refine this general approach to derive a condition table using multiple sources of information where a column in the condition table represents a test case, which is a combination of conditions to be tested [GG75]. In the category-partition method of Ostrand and Balcer, the tester analyzes the specification and identifies separately testable functional units, categorizes each function's inputs, and then partitions categories into equivalence classes [OB88]. These approaches leave test case selection completely to the tester through document reading activities.

Several researchers propose techniques that focus on selecting test cases from the specification. Weyuker and Ostrand propose revealing subdomains constructed by subdividing path domains based on likely errors, which may be derived from the specification [WO80]. Richardson and Clarke propose the partition analysis method, which develops a partition by overlaying an implementation-based partition and a specification-based partition [RC81, RC85a]. Howden's functional testing employs specification and design information for functional decomposition and applies guidelines for different functional classes to select test cases [How86]. Bouge, Chouquet, Fribourg, Gaudel and Marre present an approach for selecting descriptions of monotonically increasing collections of test sets for abstract data types from algebraic specifications [BCFG86, GM88]. Gopal and Budd propose a test adequacy technique based on mutation of a predicate calculus specification [GB83]. None of these test case selection approaches have been sufficiently well-defined to be generally applicable.

Some techniques are directed toward testing the specifications rather than the implementation. Kemmerer proposes two methods of testing functional specifications based on InaJo: symbolic execution of the specification and rapid prototyping by transformation to a procedural form [Kem85]. Goguen and Tardo support testing of algebraic specifications with OBJ [GT79]. Neither specification testing technique focuses on selecting the actual test cases for the specification.

Several approaches use the specification as an oracle and provide debugging capabilities but do not address test case selection. Gannon, McMullin and Hamlet describe DAISTS (Data Abstraction, Implementation, Specification and Testing System), which provides facilities for testing abstract data type implementations against algebraic specifications with user-supplied data [GMH81]. The Anna tool set provides capabilities for writing assertions to be compiled into run-time checks for use in a debugging methodology [LvH85]. Velasco presents a method that uses programmer-supplied assertions to select test data and detect inconsistencies in the code [Vel87].

4 New Specification-Based Testing Approaches

Our overall approach to specification-based testing is to extend implementation-based techniques to be applicable with formal specification languages and to provide a testing methodology that combines implementation-based and specification-based testing. We are defining test case selection criteria that include formal error detection and fault detection criteria. We also suggest the "active" use of the specification as an oracle to be violated, because error detection is more likely when the specification is not satisfied. We focus on test case selection based on these ideas.

A *test case* consists of an input criterion and an acceptance criterion. The input criterion is a condition describing data that satisfies this test case; it may be as specific as actual test data or as general as a condition on the input domain or output range. The acceptance criterion is a condition describing whether or not execution of this test case is acceptable or whether an error has been revealed. In some cases, the acceptance criterion is an output description, which may specify expected output values, an expected computation in terms of the inputs, or an output assertion. In others, the acceptance criterion may be a human oracle, which we denote here as "*ok?*". We present several examples of test case descriptions below.

A structural coverage technique requires a set of paths to be executed by the test cases. A test case might consist of the path condition, which describes the path domain, where the path representation is derived by symbolic evaluation [RC85b], and a human oracle applied to the actual output.

structural coverage test case	
input:	path condition
accept:	*ok?*(actual output)

On the other hand, a *specification coverage* technique for a pre/post-condition language, such as Larch's interface language, would require a test case for each pre/post-condition pair.

pre/post-condition coverage test case	
input:	pre-condition
accept:	post-condition

For some approaches, the mere fact that a test case exists satisfying the input description implies that an error has been detected; hence, the acceptance criterion is *false*. One such example is a case selected by attempting to violate a pre/post-condition pair.

pre/post-condition violation test case	
input:	pre-condition and $\neg$post-condition
accept:	*false*

A test case description may cover only a portion of the specification or implementation. For instance, use of an assertion language, like Anna, may produce intermediate assertions, and acceptance of a test run must be evaluated at the appropriate location. Two useful locations are *pre*, indicating before execution, and *post*, indicating after execution.

intermediate assertion test case	
input:	$\text{assertion}_{\text{pre}}$
accept:	$\text{assertion}_{\text{post}}$

The specification-based testing approaches discussed below select test cases in the described form. We describe these approaches as they apply to pre/post-condition pairs. Although it may not be obvious, specifications in most languages can be represented in this form. For instance, symbolic evaluation of a procedural specification provides path conditions and path values which serve as pre-conditions and post-conditions, respectively.

This representation of test cases facilitates three testing methods. First, the test cases can be used as test adequacy metrics. The test data selected (by some other means) can be checked to determine which required test cases have not yet been tested. Second, the test cases can be used for test data selection. The input criterion can be solved to provide test data. Third, the test cases can be used as a test oracle. The actual output produced for a test datum satisfying a test case input criterion can be compared against the corresponding acceptance criterion.

4.1 Specification/Error-Based Testing

Spec/error-based testing attempts to detect errors in the implementation that derive from misunderstanding the specification or errors in the specification. Error-based testing techniques are typically based on analysis of a symbolic representation of the implementation, but the ideas can be extended for application to most formal specification languages. Symbolic evaluation of a specification partitions the input space in much the same way as program paths partition the implementation domain. The form of the specification partition depends on the type of specification language. Evaluation of a pre/post-condition language, such as Larch's interface language, partitions the domain by the pre-conditions. The representation derived by symbolic evaluation is symbolic pre/post-condition pairs. For an assertion language, such as Anna, ordered pairs of assertions form pre/post-condition pairs. Evaluation of an algebraic specification language, such as Larch's shared language, or a state-based language, such as InaJo, selects specification "paths" partitioned by the procedural constructs; each path domain/computation pair defines a pre/post-condition pair.

The general approach is to select test cases that would detect potential errors in the representation. We apply the error-based techniques described above to the pre/post-condition representation. The pre-conditions and post-conditions are not distinguished between domain and computation as neatly as are path domain/computation. Thus, we apply each technique to both representations as appropriate. The domain testing technique drives the selection of boundary values of the domain specified by the pre-conditions and post-conditions.[5]

spec/domain test case for pre-condition	
input:	*boundary*(pre)
accept:	($\forall$ i $\text{pre}_i \Rightarrow \text{post}_i$) and *ok?*

spec/domain test case for post-condition	
input:	pre and *boundary*(post)
accept:	post and *ok?*

The computation testing technique drives the selection of special values of the computations specified by the pre-conditions and post-conditions.

spec/computation test case for pre-condition	
input:	*special*(pre)
accept:	($\forall$ i $\text{pre}_i \Rightarrow \text{post}_i$) and *ok?*

spec/computation test case for post-condition	
input:	pre and *special*(post)
accept:	post and *ok?*

4.2 Specification/Fault-Based Testing

The goal of spec/fault-based testing is to detect faults in the specification by revealing specification failures or to detect coding faults that are due to misunderstanding the specification by revealing implementation failures. Fault-based testing techniques postulate that faults exist in the implementation and select test cases to detect those faults if they exist. These techniques are extended to be applied to formal specifications. The fault classes are, of course, dependent on the specification language. The general approach is to select test cases that would detect potential faults in the specification source. Even if the specification is correct, these hypothesized faults may still be indicative of faults that might be introduced in the implementation. We employ the RELAY model to select revealing conditions that distinguish the variant from the source and apply it to both pre-conditions and post-conditions.

spec/fault-based test case for pre-condition	
input:	*revealing*(*variant*(pre) $\neq$ pre)
accept:	($\forall$ i $\text{pre}_i \Rightarrow \text{post}_i$) and *ok?*

[5] pre/post-condition pairs are subscripted only when their association is not obvious.

spec/fault-based test case for post-condition	
input:	pre and *revealing*(*variant*(post) $\neq$ post)
accept:	post and *ok?*

4.3 Oracle/Error-Based Testing

Oracle/error-based testing applies error-based techniques to the implementation while explicitly attempting to force a violation of the oracle as embodied in the specification. Domain testing techniques are applied to select boundary values of the path domain as described by the path condition (PC), and computation testing techniques are applied to select special values of the path computation as described by the path values (PV).

oracle/domain test case for PC	
input:	*boundary*(PC) and pre and $\neg$post
accept:	*false*

oracle/computation test case for PV	
input:	*special*(PV) and pre and $\neg$post
accept:	*false*

Another form of oracle/error-based testing is incompleteness testing, where test cases are selected for portions of the input domain left unspecified. A violation of the oracle occurs if none of the pre-conditions are satisfiable. This does not necessarily indicate an error as the specification may be indifferent for some input or simply incomplete, but supports the tester in determining whether the implementation handles these cases properly.

oracle/incompleteness test case	
input:	$(\forall i \neg \text{pre}_i)$
accept:	*ok?*

4.4 Oracle/Fault-Based Testing

Oracle/fault-based testing focuses on detecting specific faults in the implementation by transferring resulting errors to violate the specification. We again employ the RELAY model in this context. For a potential fault, we select the revealing condition up to a post condition that references an error and attempt to force one of the source or variant to satisfy the post condition and the other to violate it. If a post condition is violated, a fault has been detected and we have cut the transfer route. Otherwise, extension of the transfer route is required, as defined by the model. In the extreme, transfer to output may be required when no post condition can be violated.

oracle/fault-based test case	
input:	(PC and *revealing*(*variant*(source) $\neq$ source) and (*variant*(post) $\neq$ post)
accept:	post and *ok?*

5 Specification-Based Testing with Anna

5.1 Overview of Anna

ANNA is a specification language designed to extend the Ada programming language. As such, it is perhaps closer to a design language than a specification language and its use risks biasing the implementation. This aside, ANNA presents an interesting vehicle for specification-based testing. The intent of ANNA, as described by Luckham and von Henke, is to support the "activity of explanation" by making programs more readable and facilitating program design [LvH85]. We provide an overview of the ANNA language here; more thorough descriptions appear elsewhere [LvH85, L$^+$84].

```
package IntStack is

  type Stack_Type is private;

  Max_Elems: constant INTEGER := 100;
  Overflow, Underflow: exception;

  --: function Length (Stack: Stack_Type) return INTEGER;

  function Empty (Stack: Stack_Type) return BOOLEAN;

  function Full (Stack: Stack_Type) return BOOLEAN;

  function Top (Stack: Stack_Type) return INTEGER;

  function Create return Stack_Type;

  procedure Push (Stack: in out Stack_Type; Item: INTEGER);
  --| where Full(in Stack) => raise Overflow,
  --|       raise Overflow => Stack = in Stack,
  --|       out(Length(Stack) = Length (in Stack)+1),
  --|       Top(Stack) = Item;

  procedure Pop (Stack: in out Stack_Type; Item: out INTEGER);
  --| where Empty(in Stack) => raise Underflow,
  --|       raise Underflow => Stack = in Stack,
  --|       out(Length(Stack) = Length(in Stack)-1),
  --|       Top(in Stack) = out Item;

  --: function Push_Pop (S: Stack_Type) return Stack_Type;
  -- The function returns a stack that is the result of pushing
  -- a random element on the stack and then popping it off.

  --| axiom for all S:Stack_Type => S = Push_Pop(S);

private

  subtype Stack_Range is INTEGER range 0..Max_Elems;
  type Elem_List is array (1..Max_Elems) of INTEGER;
  type Stack_Type is
    record
      Count: Stack_Range;
      Elems: Elem_List;
    end record;
end IntStack;
```

Figure 2: ANNA/Ada Package Specification of Stack

An ANNA specification consists of assertions in Ada code. Without Ada source code, the ANNA constructs stand as an independent specification that may be further developed. Assertions are written in ANNA constructs for virtual text, quantified expressions, and annotations. The assertions do not affect the Ada code, as they are within comments and are restricted from altering the values of actual objects in the code. Figures 2 and 3 present ANNA/Ada specification and implementation of a stack package, while Figure 4 presents an ANNA/Ada implementation of a Square Root function. These examples will be used to illustrate both ANNA's syntax and its utility for testing.

5.1.1 Virtual Text

Virtual text is simply Ada-style code that can refer to, but not change, actual objects. The main purpose of virtual code is to define functions that will be used in the annotations and clarify the intended meaning of the code. In the stack example, the virtual function Length, which appears in both specification and body, is used to explain the semantics for Push and Pop. Push_Pop appears in quantified expressions

```
package body IntStack is

   --: function Length (Stack: Stack_Type) return INTEGER is
   --: begin
   --:    return Stack.Count;
   --: end Length;

   function Empty (Stack: Stack_Type) return BOOLEAN is
   begin
      return Stack.Count = Elem_List'FIRST;
   end Empty;

   function Full (Stack: Stack_Type) return BOOLEAN is
   begin
      return Stack.Count = Elem_List'LAST+1;
   end Full;

   function Top (Stack: Stack_Type) return INTEGER is
   begin
      return Stack.Elems(Stack.Count);
   end Top;

   function Create return Stack_Type is
   Stack: Stack_Type := (Elem_List'FIRST, (others => 0));
   begin
      return Stack;
   end Create;

   procedure Push (Stack: in out Stack_Type;
                          Item: INTEGER) is
   begin
      if not Full(Stack) then
         Stack.Count := Stack.Count + 1;
         Stack.Elems(Stack.Count) := Item;
      else
         raise Overflow;
      end if;
   end Push;

   procedure Pop (Stack: in out Stack_Type;
                         Item: out INTEGER) is
   begin
      if not Empty(Stack) then
         Item := Stack.Elems(Stack.Count);
         Stack.Count := Stack.Count - 1;
      else
         raise Underflow;
      end if;
   end Pop;

end IntStack;
```

Figure 3: ANNA/Ada Package Body of Stack

to describe the consistent behavior of the stack.

5.1.2 Quantified Expressions

Quantified expressions in ANNA are more general than in classical first order logic since ANNA quantified expressions may not always have defined values. A quantified expression, for example, **for all** X:T => P(X), has a defined value even if there exist some values in T for which P(X) is undefined. It is true as long as there is no value of X for which P(X) is both defined and false; false otherwise. If there are no undefined values for P(X) then the quantified expression has the same definition as that of classical logic.

An example of a quantified expression in the stack example appears on line 34, **for all** S:Stack_Type => S = Push_Pop(S) , where Push_Pop is a virtual routine that checks that every newly inserted item goes on top of the stack.

5.1.3 Annotations

Annotations are Boolean expressions that denote conditions that must hold true over some region of code, be it a single statement or the entire program. ANNA provides a rich set of annotations, each varying in their scope of influence.

All annotations may make use of virtual text, logical variables, and actual variables. Some types of annotations may make use of a variable's values before calculation and after, particularly in the case of annotations concerning preconditions and post-conditions on subprograms. To distinguish between a variable's original and current values, the key words **in** and **out** are used.

There are two types of statement annotations: a *simple statement annotation* constrains the state after execution of the preceding statement and a *compound statement annotation* constrains the execution of the succeeding compound statement. The annotation on line 10 in Square_Root is a simple statement annotation that must hold after execution of the statement at line 20.

An *object annotation* is a condition associated with a specific object and is equivalent to placing simple statement annotations after every reference to the object. Likewise, a *type annotation* is a condition on types and applies to all objects of that type. The assertions on lines 4 and 5 of Square_Root are object annotations on High and Low, respectively. These two annotations also imply restrictions on Mid.

A *subprogram annotation* provides a means of listing preconditions and post-conditions for a subprogram. These annotations are illustrated in the stack specification, where both Push and Pop have conditions they must meet. The subprogram annotations at lines 21 and 22 indicate that Push must increment Stack's size and store Item on the top of Stack. Lines 26 and 27 indicate that Pop must decrement Stack's size and return what was on the top of Stack as Item. Although they are not complete, these annotations explain the last-in-first-out behavior of stacks, without saying how package Stacks is implemented.

Exception annotations come in two varieties. A *weak exception annotation* describes the conditions that will be true if an exception is raised. A *strong exception annotation* describes those conditions under which exceptions *must* be raised. Both are illustrated in the stack specification. In Pop, the strong exception annotation at line 25 states when the incoming Stack is full, the Overflow exception must be raised. In addition, the weak exception annotation at line 26 says if the exception Overflow is raised, Stack will remain unchanged. Push has similar exception annotations.

Context annotations simply expand on the concept of with and use clauses and can be statically checked at compile time. Therefore, they would be useful in integration testing; we do not consider them here.

ANNA's *package axioms* are used to define the properties of the allowable operations on a private type by algebraic specifications. This allows a powerful, implementation-independent method of specifying abstract data types. The axiom at line 34 in the stack specification defines the LIFO behavior of stacks.[6]

Package states are a method of specifying a trace of the actions performed with the package operations. The package itself is regarded as having its own type. The initial value, or initial state, of the package type is the trace where no subprograms of the package have yet been invoked. A useful state might refer to a sequence of calls to Push and Pop, particularly for defining an axiom that describes the

[6]The virtual function Push_Pop is defined to simulate the action of Pop(Push(Stack, Elem)). This cannot be done directly, since Ada does not allow modification of function parameters.

```
function SquareRoot (N: NATURAL) return NATURAL is
  --| where return S:NATURAL => S**2 <= N < (S+1)**2;

    Low  : NATURAL := 1;        --| Low <= Mid;
    High : NATURAL := N/2+1;   --| High >= Mid;
    Mid  : NATURAL := (Low+High)/2;

  begin
    loop
      --| Low**2 <= N <= High**2;                                  10
      exit when (Mid**2 <= N) and ((Mid+1)**2 > N);
      if Mid**2 > N then
        High := Mid;
      elsif Low = Mid then
        Low := Low+1;
      else
        Low := Mid;
      end if;
      Mid := (Low+High)/2;
    end loop;                                                      20
    return Mid;
  end SquareRoot;
```

Figure 4: ANNA/Ada Implementation of Square Root

behavior of a push followed by a pop.

5.2 Specification/Error-Based Testing

Suppose that we select domain test cases for the object annotation Low $\leq$ Mid in SquareRoot (Figure 4, line 5). This annotation along with the object annotation High $\geq$ Mid are the pre-conditions and S**2 $\leq$ N $<$ (S+1)**2 is the post-condition. Symbolic evaluation of the pre-condition and post-condition pair based on symbolic input for N derives the following pre and post condition pair:

pre:	(N_{pre}/2+1 $\geq$ ((N_{pre}/2+1)+1)/2) and (1 $\leq$((N_{pre}/2+1)+1)/2)
post:	($Low_{post} \leq Mid_{post}$) and ($High_{post} \geq Mid_{post}$) and ($Mid_{post}$**2 $\leq$ N $<$ (Mid_{post}+1)**2)

Selecting boundary values for the pre-condition 1 $\leq$ ((N_{pre}/2+1)+1)/2 would select the following test case:

input:	(N_{pre}/2+1 $\geq$ ((N_{pre}/2+1)+1)/2) and (1=((N_{pre}/2+1)+1)/2)
accept:	($High_{post} \geq Mid_{post}$) and ($Low_{post} \leq Mid_{post}$) and ($Mid_{post}$**2 $\leq$ N $<$ (Mid_{post}+1)**2) and *ok?*

N = 0 satisfies this input condition and execution violates the loop assertion. This test case focuses attention onto Low, and we determine that Low should be initialized to 0.

For the following testing approaches, we will assume that the initialization of Low has been fixed to be zero.

5.3 Specification/Fault-Based Testing

In applying spec/fault-based testing to SquareRoot, suppose we posit that the annotation High $\geq$ Mid should be High = Mid (Figure 4, line 5). A partial revealing condition is:

PC:	*true*
origination condition:	n/2 + 1 = (n/2 + 1)/2

The origination condition is infeasible and thus transfer need not be considered. Since the revealing condition can not be satisfied, we are assured that this variant is not a fault (the variant module is equivalent to the original). The original is redundant (that is, it need only be High $>$ Mid) since it is not possible to satisfy High = Mid.

5.4 Oracle/Error-Based Testing

If we apply oracle/error-based testing to Square_Root and symbolically evaluate the path up to the assignment to Low := Low+1 (Figure 4, line 15), the following path representation is derived:

PC:	(((N/2+1)/2)**2 $>$ N or ((N/2+1)/2+1)**2 $\leq$ N) and (((N/2+1)/2+1)**2 $\leq$ N) and (0=(N/2+1)/2)
PV:	N=N, Low=1, High = N/2+1, Mid = (N/2+1)/2

At the assignment to Low that follows, suppose we attempt to violate the object annotation on Low. Thus, we select the following test case:

input:	(((N/2+1)/2)**2 $\leq$ N) and (0=(N/2+1)/2) and (1 $>$ (N/2+1)/2)
accept:	*false*

The value N=1 satisfies this input condition. This reveals the inconsistency between the annotation Low $\leq$ Mid and the assignment Low := Low+1 in the else branch of line 14. The object annotation on Low should be Low $\leq$ Mid+1.

5.5 Oracle/Fault-Based Testing

Suppose we hypothesize that High := N/2+1 in SquareRoot should be High := (N+1)/2 (Figure 4, line 5). The revealing condition is:

PC:	*true*
origination condition:	OC = N/2+1 $\neq$ (N+1)/2
transfer condition:	TC = ((N $\leq$ (N/2+1)**2) and (N $>$ ((N+1)/2)**2)) or ((N $>$ (N/2+1)**2) and (N $\leq$ ((N+1)/2)**2))

Therefore we have the test case:

input:	OC and TC
accept:	((Low**2 $>$ N) or (N $>$ High**2)) and *ok?*

N = 2 satisfies this input condition and causes a violation of the loop assertion for the proposed variant, yet the original source does not violate the assertion. Further inspection reveals that both variants run correctly, however, so the right side of the loop assertion comes under suspicion and is corrected to become (High+1)**2.

6 Specification-Based Testing with Larch

6.1 Overview of Larch

Larch is a two-tiered specification language [Win83, GHW85]: the shared language tier and the interface language tier. The shared language specification defines an implementation language independent theory for the abstract data type, while the language specific interface language describes the module interfaces based on that theory. Since the shared language specification is language-independent, it can be written before the implementation language is known and can be reused, even for programs in different languages. On the other hand, the interface language specifies the module's implementation level interface.

```
STACKOFINT: trait
    imports INTEGER
    introduces
        EMPTY: → INTSTACK
        PUSH: INTSTACK x INTEGER → INTSTACK
        TOP: INTSTACK → INTEGER
        REST: INTSTACK → INTSTACK
        SIZE: INTSTACK → INTEGER
    closes INTSTACK over [NEW,PUSH]
    partitioned by [TOP,REST]
    constrains [INTSTACK] for all [S:INTSTACK;I:INTEGER]
        REST(PUSH(S,I)) = S
        REST(EMPTY) EXEMPT
        TOP(PUSH(S,I)) = I
        TOP(EMPTY) = ERROR
        SIZE(EMPTY) = 0
        SIZE(PUSH(S,I)) = 1 + SIZE(S)
```

Figure 5: Larch Shared Language Specification of Stack

6.1.1 Larch Shared Language

Larch's shared language extends the capabilities of algebraic specification. The extensions do not increase the power of the algebraic specifications, but provide more information that can be used for consistency checking. In the shared language, **traits**, **sorts** and **terms** correspond to modules, types and objects, respectively, in the implementation language. Figure 5 provides a shared language **trait** defining a stack of integers.

A trait may **import** or **include** previously defined traits. In either case, the new trait may rely on the properties of the previously-defined trait. An imported trait cannot be affected by the axioms of the new trait, but an included trait can be extended or constrained by the new trait. **Import** and **include** clauses allow inheritance and modularization between traits to facilitate information hiding. In Figure 5, the trait INTEGER is imported to define a trait STACKOFINT.[7]

The **closes** clause guarantees that any term of the sort which is defined by the **trait** can be made with a sequence of the specified operations. In STACKOFINT, the **closes** clause indicates that any stack can be made from sequences of PUSH onto an EMPTY stack. The **partitioned by** clause means that two **terms** of the **sort** are equal if and only if all of the functions mentioned in the **partitioned by** clause are equal. 'TOP' and 'REST' partition stacks; if TOP and REST are equal for two stacks, the stacks are equal. Finally, the **exempt** clause states that the results have been left intentionally unspecified.

6.1.2 Larch Interface Language

Larch's family of interface languages provides interfaces between the theory of the shared language specification and implementations. Different implementation languages have different properties, and hence have different interface languages. For example, the CLU interface language provides

[7]For the sake of clarity, all of the shared language identifiers in this paper will be upper case, while the interface language identifiers will be in mixed case.

```
package IntStack is
    provides mutable Stack from STACKOFINT.INTSTACK

    function Empty(S:Stack) returns Boolean is
        pre: true
        post Empty = (SIZE(S) = 0)
    end;

    function Full(S:Stack) returns Boolean is
        pre: true
        post: Full = (SIZE(S) = 100)
    end;

    function Top(S:Stack) returns Integer is
        pre: true
        post: Top = TOP(S)
    end;

    function Create return Stack is
        pre: true
        post: Create = EMPTY and new(Create)
    end;

    procedure Push (S: in out Stack; E: Integer) is
        pre: SIZE(S) ≤ 100
        mutates: S
        post: S_post = PUSH(S_pre,E)
    end;

    procedure Pop (S: in out Stack; I: out Integer) is
        pre: SIZE(S) > 0
        mutates: S
        post: I_post = TOP(S_pre) and S_post = REST(S_pre)
        pre: SIZE(S) = 0
        post: raise Stackempty
    end;
end;
```

Figure 6: Larch Interface Specification of Stack

for signals, clusters and iterators, while the Pascal one does not. On the other hand, Pascal's interface language provides the semantics of var parameters, which are not present in CLU. A design goal for each interface languages is that they approximate the syntax of the implementation language. For the examples in this paper, we have used an Ada interface language.[8] Figure 6 provides an interface specification IntStack for the previous STACKOFINT example.

To limit the visibility between the interface language and the shared language, a **provides** clause links the package to a set of the shared language traits that are visible. Each procedure or function in the package implementation is specified by a list of pre/post-condition pairs. For each pre/post pair, Pre {Function} Post, if Pre is true before the function is executed then Post must be true afterwards. These conditions can be arbitrary first order predicates, including quantifiers, that refer to the shared language component and the values of variables. Since the conditions are arbitrary, the specifi-

[8]based on the CLU and Pascal interface languages, which have been described in previous papers.

```
package IntStack is
      provides mutable Stack from stackofint.intstack
      [Rest,Top] implement [REST,TOP] – Top is already defined, Rest
must be added.
      ...
      function Rest(S:Stack) returns Stack is
            pre: S ≠ EMPTY
            post: Rest = REST(S)
      end Rest;
end IntStack;
```

Figure 7: Mapping Function Between Concrete and Abstract Values

cations can be incomplete or non-deterministic. A **mutates** clause in the middle of a pre/post pair indicates that only those parameters or global variables that can be modified by the function. The post-conditions can reference the values of the parameters or global variables before or after the function. X_{pre} is used to denote the value of the parameter X in the state before the function was called, while X_{post} is the value after the function is called.

6.1.3 Evaluating Larch Specifications in an Implementation State

When predicates in Larch specifications refer to a variable, they refer to the variable's abstract value rather than its concrete value. For example, the pre-condition for Push is SIZE(S) $\leq$ 100. Since SIZE is never defined for concrete values, only the abstract value of S can be used. To determine whether a particular state in the execution of the implementation satisfies a Larch assertion, a mapping between concrete values of the implementation and abstract values of the shared language specification is required. This is similar to the work of Gannon, Hamlet and Mills in verifying the correctness of modules from specifications[GHM87].

The mapping between the abstract and concrete values is computed by mappings from each function in the trait's **partitioned by** clause to an implemented function in the interface specification. Unfortunately, this extension to Larch may require the programmer to implement several unnecessary functions. For the stack example, the implementor would need to implement equivalent functions for TOP and REST, if they were not already implemented. In Figure 7, the interface language specification for the stack is extended with the mapping information.

By using the related abstract and concrete functions and recursing through the data structure, it is possible to find the conditions for equivalence of concrete and abstract values. In the stack, the concrete value would be broken up with Top and Rest, and the algorithm would recursively break each of the results down, until it had only atomic pieces. Figure 8 gives an example of this process. If the shared language rules for the functions in the **partitioned by** clause are complete, then there is only one solution to the selected conditions.

To use a Larch specification as an oracle, the concrete inputs and outputs must be converted into their abstract representations. If the input/output pair satisfies the pre/post-conditions, then the oracle would accept the test, otherwise it would reject it. Since both the input and output abstract representations are available, the non-determinism and incompleteness features of the interface language and exempt clauses in the shared language trait are not a problem.

Concrete Value:	(1,2,3)
Conditions:	top(S1) = 1 rest(S1) = S2 top(S2) = 2 rest(S2) = S3 top(S3) = 3 rest(S3) = error
Abstract Value:	S1 = push(1,push(2,push(3,empty)))

Figure 8: Solving for the abstract value.

```
function SquareRoot(N:integer) returns Integer is
      pre: N ≥ 0
      post: (SquareRoot ** 2 ≤ N) and (N < (SquareRoot+1) ** 2)
      pre: N < 0
      post: raise Domain_Error
end SquareRoot;
```

Figure 9: Larch Interface Specification of SquareRoot

6.2 Specification/Error-Based Testing

One application of spec/error-based testing is domain testing of Larch pre/post-conditions. To compute a value on the boundary of a pre/post condition, we recursively analyze each expression in the condition. If the operator is *and*, remove each subexpression one at a time and select a test case that violates the removed condition and a test case that satisfies it. If it is impossible to violate the removed condition, then the condition was not necessary. If the operator is *or*, then select a test case that will cause each subexpression to be true, while the others are false. An additional test case where all of the subexpressions are true should be tested. If any test cases are infeasible, the specification is incomplete and should be analyzed further with incompleteness testing. Several test case descriptions are given below as examples of this approach applied to SquareRoot (Figure 9).

input:	(N < 0)
accept:	((N ≥ 0) and (SquareRoot ** 2 ≤ N) and (N < (SquareRoot+1) **2)) or ((N < 0) and (raise Domain_Error)) and *ok?*

input:	(N ≥ 0) and (SquareRoot ** 2 > N) and (N < (SquareRoot+1) ** 2)
accept:	(SquareRoot ** 2 ≤ N) and (N < (SquareRoot+1) ** 2)

input:	(N ≥ 0) and (SquareRoot ** 2 ≤ N) and (N ≥ (SquareRoot+1) ** 2)
accept:	(SquareRoot ** 2 ≤ N) and (N < (SquareRoot+1) ** 2)

6.3 Specification/Fault-Based Testing

By spec/fault-based testing, the tester can ensure that specific faults are not in the specification or implementation. Since Larch has two tiers, faults can be hypothesized in either tier. Here we consider a fault in the interface specification. The hypothesized fault is that the post-condition for Pop in SquareRoot (Figure 6, line 33) is (I_{post} = TOP(REST(S_{pre})) and S_{post} = REST(S_{pre})) instead of (I_{post} = TOP(S_{pre}) and S_{post} = REST(S_{pre})). The following revealing condition distinguishes between the alternatives.

origination condition:	REST(S_{pre}) $\neq$ S_{pre}
transfer condition:	TOP(REST(S_{pre})) $\neq$ TOP(S_{pre})

input:	SIZE(S_{pre}) > 0 and TOP(REST(S_{pre})) $\neq$ TOP(S_{pre})
accept:	(I_{post} = TOP(S_{pre}) and S_{post} = REST(S_{pre})) and *ok?*

If such a test case can be found, then it differentiates between the specification and the variant. This test case may reveal a fault in the specification or, alternatively, may reveal a fault in the implementation (such as decrementing count before retrieving the element rather than afterwards) that is caused by misunderstanding the specification

6.4 Oracle/Error-Based Testing

Here we apply oracle/error-based testing to the corrected Ada code for SquareRoot (Figure 4) and the Larch specification (Figure 9). As an example, pick the following path (8,11,12,13,11,12,13,11,20,21) for which the following symbolic representation is derived:

PC:	$(\frac{(N+1)^2}{16} > N)$ and $(\frac{(N+1)^2}{64} > N)$ and $(\frac{(N+1)^2}{256} \leq N)$ and $(\frac{(N+17)^2}{256} > N)$
PV:	return = $\frac{(N+1)}{16}$

The symbolic values are used to evaluate the post-condition from the specifications. To violate the oracle, we must find a solution to the negation of the post-condition that can be satisfied within the restriction of the path condition.

input:	$(\frac{(N+1)^2}{16} > N)$ and $(\frac{(N+1)^2}{64} > N)$ and $(\frac{(N+1)^2}{256} \leq N)$ and $(\frac{(N+17)^2}{256} > N)$ and $(N \geq 0)$ and $((\frac{(N+1)^2}{256} > N)$ or $(\frac{(N+17)^2}{256} \geq N))$
accept:	*false*

Clearly, there are no values of N that satisfy the input condition, and therefore there are no violations of the specification along the chosen path.

Another approach for oracle/error-based testing is incompleteness testing. In Larch, incompleteness is introduced by **exempt** clauses in the shared language specification, which represent **terms** that do not have a specified value in the trait's theory, or missing cases, which are input ranges that violate all of the pre-conditions for the function. In the Larch stack specification (Figure 5), there are two ambiguities that are caused by REST(EMPTY) being exempt and pushing onto stacks of more than 100 elements. Both of the following test cases should be executed to be sure that the implementation's output is acceptable.

Test case for REST(S)	
input:	S = empty
accept:	*ok?*

Test case for PUSH(I,S)	
input:	size(S) > 100
accept:	*ok?*

6.5 Oracle/Fault-Based Testing

Suppose we hypothesize that Stack.Elems(Stack.Count) should be Stack.Elems(Stack.Count+k) in Push (Figure 3, line 34). The RELAY model can be used to reveal this potential fault by transferring an error to the post condition so that one of the variants satisfies the specification and the other does not. In other words, so that *variant*($Stack_{post}$) $\neq$ $Stack_{post}$.

Because of the **partitioned by** clause in the stack specification (Figure 5), equality between two stacks can be divided into a test of TOP and REST. A sufficient, but not necessary, condition can be derived from comparing the tops of the stacks. Therefore, if a fault-based method can transfer the incorrect state through TOP(PUSH(STACK,ITEM)) one of the variants will not satisfy the specification. Relay selects the following conditions to transfer the incorrect state through top and push.

PC:	Stack.Count $\leq$ Elem_List'Last
origination condition:	$k \neq 0$
transfer condition:	(Stack.Elems(Stack.Count+1) $\neq$ Item)

A corresponding test case description is:

input:	(Stack.Count $\leq$ Elem_List'Last) and (Stack.ELems(Stack.Count+1) $\neq$ Item) and *variant*($Stack_{post}$) $\neq$ $Stack_{post}$
accept:	*ok?* and $Stack_{post}$ = PUSH($Stack_{pre}$,Elem)

If each element of $Stack_{pre}$ has a different value than Item and the test case executes correctly, the hypothesized fault is not present.

7 Conclusion

In this paper, we describe our research in formalizing test case selection from specifications. We extend the notions of error-based and fault-based testing to provide *spec/error-based testing* and *spec/fault-based testing*. We also augment the implementation-based techniques to actively use specifications as oracles in *oracle/error-based testing* and *oracle/fault-based testing*. We then apply the techniques to two specification languages: ANNA and Larch.

We are exploring how implementation-based testing techniques may be used with formal specifications. We are continuing to examine many specification languages for which these ideas apply. InaJo [Kem85] and Refine [Rea89] are our next likely candidates. This serves not only to demonstrate the broad applicability of specification-based testing but also to help us choose one language on which to focus further efforts. We must formalize these specification-based approaches for test case selection which requires settling (for the time being) on a single semantic definition. We must also gain a solid understanding of the error detection capabilities of each approach and evaluate their strengths and weaknesses.

We are focusing now on the development of testing tools for the ANNA specification language, as part of the TEAM effort [CRZ88]. TEAM is a framework for developing tools that contain generic analysis capabilities designed to be language independent. We are currently writing a front-end for Anna that will translate it into TEAM's common internal representation, and developing symbolic evaluation capabilities for Anna. This will enable us to use the test case selection capabilities in TEAM as well as automate test data generation from the test case descriptions.

We believe that specification-based testing must be further developed and should be incorporated into the software development lifecycle. This requires the use of formal specification languages in the specification and design phases. Moreover, these test cases can be used to test specifications early in the lifecycle before they permeate the design and implementation. We also intend to explore the possibility of extending specification languages to incorporate test case descriptions that the user can specify or the environment can generate automatically. We believe that developers will be less reluctant to use formal specification languages if we can demonstrate concrete advantages to be gained from their use in testing.

References

[BCFG86] L. Bougé, N. Choquet, L. Fribourg, and M.-C. Gaudel. Test set generation from algebraic specifications using logic programming. *The Journal of Systems and Software*, 6:343–360, 1986.

[CPRZ89] Lori A. Clarke, Andy Podgurski, Debra J. Richardson, and Steven J. Zeil. A formal evaluation of data flow path selection criteria. *IEEE Transactions on Software Engineering*, SE-15(11), November 1989.

[CR83] Lori A. Clarke and Debra J. Richardson. A rigorous approach to error-sensitive testing. In *Proceedings of the Sixteenth Hawaii International Conference on System Sciences*, pages 197–206, January 1983.

[CRZ88] Lori A. Clarke, Debra J. Richardson, and Steven J. Zeil. Team: A support environment for testing, evaluation, and analysis. In *Proceedings of ACM SIGSOFT '88: Third Symposium on Software Development Environments*, pages 153–162, November 1988. Appeared as *SIGPLAN Notices 24*(2) and *Software Engineering Notes 13*(5).

[DLS78] Richard DeMillo, R. J. Lipton, and F. G. Sayward. Hints on test data selection: Help for the practicing programmer. *IEEE Computer*, 11(4), April 1978.

[GB83] Ajei Gopal and Tim Budd. "Program Testing by Specification Mutation". Technical Report TR 83-17, University of Arizona, November 1983.

[GG75] John B. Goodenough and Susan L. Gerhart. Toward a theory of test data selection. *IEEE Transactions on Software Engineering*, SE-1(2):156–173, June 1975.

[GHM87] John D. Gannon, Richard G. Hamlet, and Harlan D. Mills. Theory of modules. *IEEE Transactions on Software Engineering*, SE-13(7):820–829, 1987.

[GHW85] John Guttag, James Horning, and Jeanette Wing. The larch family of specification languages. *IEEE Transactions on Software Engineering*, pages 24–36, September 1985.

[GM88] M.-C. Gaudel and B. Marre. Algebraic specifications and software testing: Theory and application. In *Rapport LRI #407*, 1988.

[GMH81] John Gannon, Paul McMullin, and Richard Hamlet. Data-abstraction implementation, specification, and testing. *ACM Transactions on Programming Languages and Systems*, 3(3):211–223, July 1981.

[Gou83] John S. Gourlay. A Mathematical Framework for the Investigation of Testing. *IEEE Transactions on Software Engineering*, SE-9(6):686–709, November 1983.

[GT79] Joseph A. Goguen and Joseph J. Tardo. An introduction to obj: A language for writing and testing formal algebraic program specifications. In *IEEE Conference on Specification of Reliable Software*, pages 170–188, 1979.

[Ham77] Richard G. Hamlet. Testing programs with the aid of a compiler. *IEEE Transactions on Software Engineering*, SE-3(4):279–290, July 1977.

[How78] William E. Howden. Introduction to the theory of testing. In Edward Miller and William E. Howden, editors, *Tutorial: Software Testing and Validation Techniques*, pages 16–19. IEEE, New York, 1978.

[How86] William E. Howden. A functional approach to program testing and analysis. *IEEE Transactions on Software Engineering*, SE-12(10):997–1005, October 1986.

[Kem85] Richard A. Kemmerer. Testing formal specifications to detect design errors. *IEEE Transactions on Software Engineering*, 11(1):32–43, January 1985.

[L+84] David C. Luckham et al. Anna: A language for annotating Ada programs. Technical Report 84-261, Stanford University, July 1984.

[Las88] Janusz Laski. Testing in top-down program development. In *Proceedings of the Second Workshop on Software Testing, Verification, and Analysis*, pages 72–79, Banff, July 1988. ACM/SIGSOFT and IEEE-CS Software Engineering Technical Committee.

[LvH85] David C. Luckham and Friedrich W. von Henke. An overview of ANNA, a specification language for Ada. *IEEE Software*, 2(2):9–22, March 1985.

[Mye79] G. J. Myers. *The Art of Software Testing*. John Wiley & Sons, New York, 1979.

[OB88] Thomas J. Ostrand and Marc J. Balcer. The category-partition method for specifying and generating functional tests. *Communications of the ACM*, 31(6):676–686, June 1988.

[OSW86] Thomas Ostrand, Ron Sigal, and Elaine Weyuker. Design for a tool to manage specification-based testing, July 1986.

[RC81] Debra J. Richardson and Lori A. Clarke. A partition analysis method to increase program reliability. In *Proceedings of the Fifth International Conference on Software Engineering*, pages 244–253, San Diego, California, March 1981.

[RC85a] Debra J. Richardson and Lori A. Clarke. Partition analysis: A method combining testing and verification. *IEEE Transactions on Software Engineering*, SE-11(12):1477–1490, December 1985.

[RC85b] Debra J. Richardson and Lori A. Clarke. Testing techniques based on symbolic evaluation. In T. Anderson, editor, *Software: Requirements, Specification and Testing*, pages 93–110. Blackwell Scientific, 1985.

[Rea89] Reasoning Systems, Palo Alto, California. Refine *User's Guide*, 1989.

[RT86] Debra J. Richardson and Margaret C. Thompson. Relay: A new model for error detection. COINS Technical Report 86–64, University of Massachusetts, Amherst, Massachusetts, December 1986.

[RT88] D. J. Richardson and M. C. Thompson. The RELAY model of error detection and its application. In *Proceedings of the Second Workshop on Software Testing, Verification, and Analysis*, pages 223–230, Banff, Canada, July 1988. ACM/SIGSOFT and IEEE-CS Software Engineering Technical Committee.

[Vel87] F. R. D. Velasco. A method for test data selection. *Journal of Systems and Software*, 7:89–97, 1987.

[WC80] Lee J. White and Edward I. Cohen. A domain strategy for computer program testing. *IEEE Transactions on Software Engineering*, SE-6(3):247–257, May 1980.

[Win83] Jeannette Wing. *A Two-Tiered Approach to Specifying Programs*. PhD thesis, MIT, 1983.

[WO80] Elaine J. Weyuker and Thomas J. Ostrand. Theories of program testing and the application of revealing subdomains. *IEEE Transactions on Software Engineering*, SE-6(3):236–246, May 1980.