Approaches to Specification-Based Testing

(Technical Report 89-19)

Debra Richardson
Owen O'Malley
Cindy Tittle

May 1989

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 that extend implementation-based testing techniques. We demonstrate these approaches for the Anna and Larch specification languages.

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 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. 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, or functional, 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 generation based on these documents. Figure 1 shows a software lifecycle model where quality assessment and test case generation are pervasive. At each phase, the generated 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 generating test cases. Specification-based test case generation has traditionally consisted of user-selected "functional" test cases based on system specifications. When the specification is informal, as is too often the case, this is effectively all that can be done. Specification-based testing systems can manage the test cases [OSW86], but automated test case generation 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 generated from formal module specifications. It is on these highlighted boxes in Figure 1 that we focus in the research described here. Unfortunately, very few such techniques exist.

In this paper, we describe our research aimed at developing approaches to specification-based test case generation. We believe that traditional functional testing can be formalized by extending implementation-based techniques to be applicable with formal specification languages. The approach we are developing is applicable to a variety of languages, including algebraic specifications, state-based specifications, pre/post conditions, assertions, and procedural design languages.

In the next section, we briefly overview past research in specification-based testing. In section three, 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. Two approaches, spec/error-based testing and

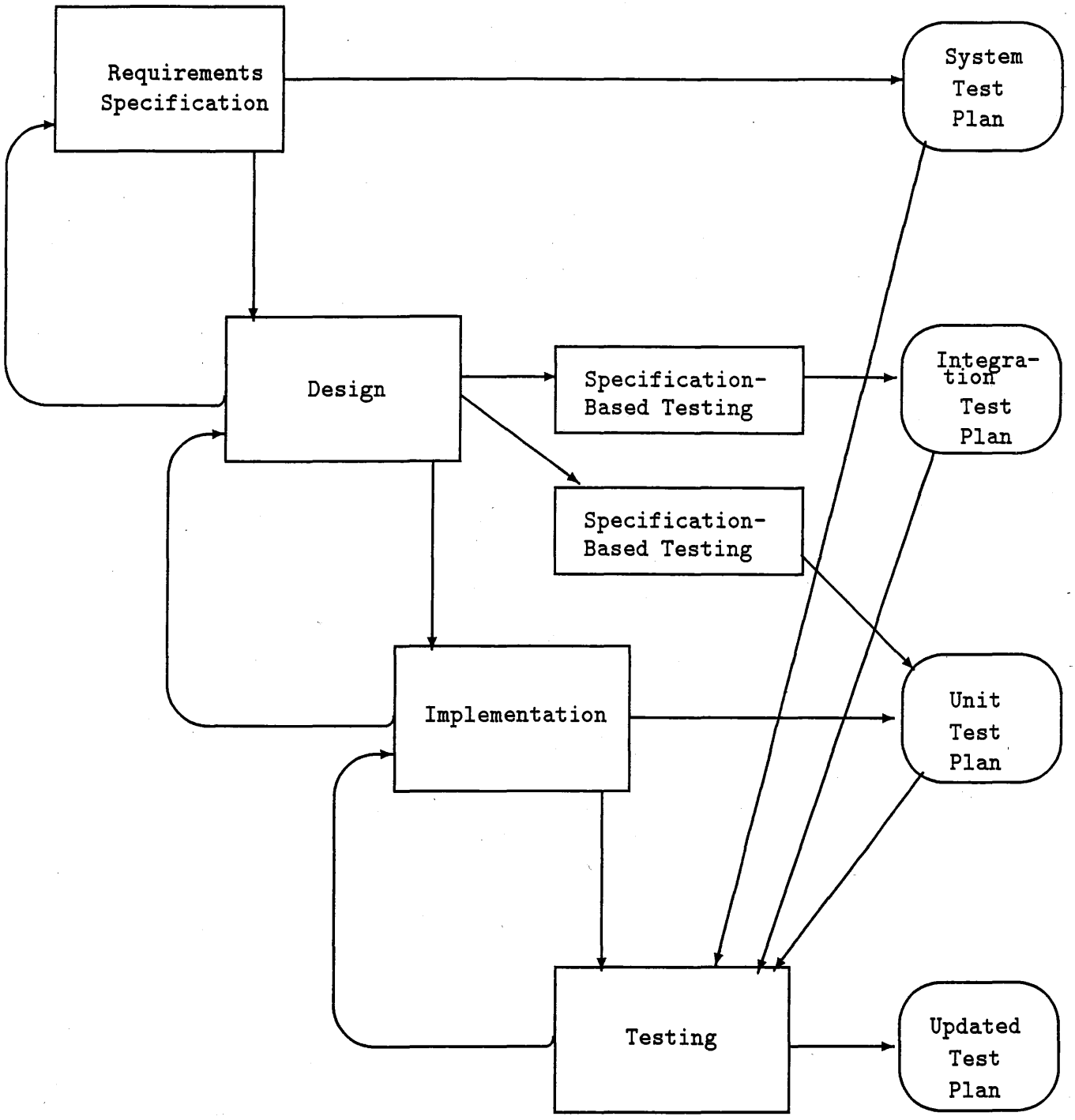

Figure 1: Test Case Generation Throughout The Software Lifecycle

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, respectively, 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, respectively, are applied to the implementation, in the hopes of detecting errors. Sections four and five apply the approaches to two specification languages: Anna and Larch. In conclusion, we discuss our intentions for future research.

2 Previous Work in Specification-Based Testing

It has long been acknowledged that test case generation 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 generation completely to the tester through document reading activities.

Several researchers propose techniques that focus on generating 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 generating descriptions of monotonically increasing collections of test sets for abstract data types from algebraic specifications[BCFG86, GM]. Gopal and Budd propose a test adequacy criteria based on mutation of a predicate calculus specification [GB83]. None of these test case generation 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 Ina Jo: symbolic execution of the specification and rapid prototyping by transformation to a proce-

dural 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 data selection. Gannon, McMullin and Hamlet discuss 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].

3 Specification-Based Testing Approaches

Our overall approach to 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 expanding traditional functional testing to 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 generation 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 criterion requires a statically-determined set of paths to be executed by the test cases. A test case might consist of the path condition and a human oracle applied to the path computation, where the path representation is derived by symbolic evaluation [RC85b].

structural coverage test case	
input:	path condition
accept:	*ok*(path computation)

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 case is generated by attempting to violate a pre/post-condition pair.

pre/post-condition violation test case	
input:	pre-condition and ¬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:	$\text{assertion}_{\text{location-i}}$
accept:	$\text{assertion}_{\text{location-a}}$

The specification-based testing approaches discussed below generate test cases in the described form. 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 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. Third, the test cases can be used for test data selection. The simplest test data selection technique would solve the input criterion to provide test data.

We are extending the ideas underlying implementation-based testing techniques to generate test cases from specifications. 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. Structural techniques typically address either paths or test data.

Path selection techniques are concerned with which statements or combinations of statements that should be executed for increased error detection. The most common path selection techniques are *control flow coverage* criteria. *Data flow coverage* is proposed as more sensitive to error detection. A survey and graph-theoretic analysis of several path selection techniques appears in [CPRZ85]. 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[1] 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 and by DeMillo, Lipton and Sayward. 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[2] provides a fault-based criterion 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. 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. We describe each of these approaches below and then provide several examples of their application to Anna and Larch in the sections that follow.

3.1 Spec/Error-Based Testing

Spec/error-based testing attempts to detect errors in the specification or errors in the implementation that are derived from misunderstanding 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

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

[2] RELAY is so named because of it analogy with a relay race.

derived by symbolic evaluation is 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, generates specification "paths" partitioned by the procedural constructs; each path domain/computation pair defines a pre/post-condition pair.

The general approach is to generate 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/post-conditions are not distinguished into domain and computation as neatly as are path domain/computation. Thus, we refer to the techniques as boundary testing and special values testing and will apply each technique to both representations as appropriate. The boundary testing technique drives the generation of boundary values of the pre-conditions and post-conditions.[3]

spec/domain test case	
input:	*boundary*(pre)
accept:	($\forall$ i $pre_i \Rightarrow post_i$) and *ok*

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

The special values testing technique drives the selection of special values of the pre-conditions and post-conditions.

spec/computation test case	
input:	*special*(pre)
accept:	($\forall$ i $pre_i \Rightarrow post_i$) and *ok*

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

3.2 Spec/Fault-Based Testing

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

spec/fault-based test case	
input:	*revealing*(*variant*(pre)) $\neq$ pre
aceept:	($\forall$ i $pre_i \Rightarrow post_i$) and *ok*

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

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

3.3 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, and computation testing techniques are applied to select special values of the path computation.

oracle/error-based test case	
input:	*boundary*(PC) and pre and ¬post
accept:	*false*

oracle/error-based test case	
input:	*special*(PV) and pre and ¬post
accept:	*false*

Another 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.

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

3.4 Oracle/Fault-Based Testing

Oracle/fault-based testing focuses on detecting specific faults in the implementation by transfer resulting errors to violate the specification. We again employ the RELAY model in this context. For a potential fault, we generate the revealing condition up to a post condition that references an error and attempt to force the source and variant to satisfy the post condition and the other to violate it. If a post condition is violated, an error has been revealed and we have cut the transfer route. Otherwise, however, extension of the transfer route is required, as defined by the model.

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

4 Specification-Based Testing with Anna

4.1 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 are given elsewhere[LvH85, LvHKBO84].

An ANNA specification consists of assertions in Ada code.[4] 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 usefulness in testing.

4.1.1 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.

4.1.2 Quantified Expressions

ANNA contains the power of classical first-order logic with the inclusion of quantified expressions. For example, from the output assertion on the Square_Root function, we may conclude that:

for all N:NATURAL => exist S:NATURAL => S <= Square_Root(N) < S+1;

The assertion above is easily expressed but requires a theorem prover to verify. On the other hand, a quantified expression such as:

for all P:Days => exist D:Days => D = Sat or D = Sun;

which indicates that of all the objects of type Days in existence, there must always exist one that has the value of either Sun or Sat. Test cases to check an expression like this one can be easily constructed.

An example of a quantified expression in the stack example is:

for all S:Stack_Type => S = Push_Pop(S);

where the function is defined as a virtual routine that checks that every newly inserted item goes on top.

4.1.3 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.

[4] Without Ada source code, the ANNA constructs stand as an independent specification.

```
package Stacks 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;                    10

  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,                          20
  --|       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;               30
  -- 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                                                    40
    record
      Count: Stack_Range;
      Elems: Elem_List;
    end record;
end Stacks;
```

Figure 2: Stack Specification in ANNA/Ada

```
package body Stacks 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;                          10
   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);                               20
   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                                                             30
      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                                                             40
      if not Empty(Stack) then
         Item := Stack.Elems(Stack.Count);
         Stack.Count := Stack.Count - 1;
      else
         raise Underflow;
      end if;
   end Pop;

end Stacks;
```

Figure 3: Stack Body in ANNA/Ada

```
function Square_Root (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 Square_Root;
```

Figure 4: Square Root Example in ANNA Ada

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 pre/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 pre/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 have 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 annotation at line 25 says that when the incoming Stack is full, the Overflow exception must be raised. In addition, the weak 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 methods. This allows a powerful, implementation-independent method of specifying abstract data types. The axiom illustrated at line 34 in the stack specification defines the LIFO behavior of stacks.[5]

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 behavior of a push followed by a pop.

4.2 Spec/Fault-Based Testing

In the application of Spec/Fault-Based testing, we posit that Low $\leq$ Mid on line 4 should be Low $\leq$ Mid + 1. The revealing condition is:

origination condition	trivial (true)
transfer condition	$1 = (n/2 + 2)/2$

The only values of n that satsify the above conditions are n=0 and n=1. The first value causes an infinite loop for both original and variant assertions, as well as violating the loop assertion in each case. The latter causes an immediate exit from the loop with the correct output for both assertions. This demonstrates that the two assertions are equivalent for this module.

4.3 Spec/Error-Based Testing

Suppose that we do boundary testing on the object annotation Low $\leq$ Mid. We can examine the symbolic evaluation of the initial values of N, obtaining the following pre/post condition

[5]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.

pair:

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

Boundary testing applied to the pre-condition 1 ≤ ((N_{pre}/2+1)+1)/2 would generate the following test case:

input	(N_{pre}/2+1 ≥ ((N_{pre}/2+1)+1)/2) and (1=((N_{pre}/2+1)+1)/2)
accept	($High_{post}$ ≥ Mid_{post}) and (Low_{post} ≤ Mid_{post}) and (Mid_{post}**2 ≤ N < (Mid_{post}+1)**2) and Okay

N = 0 satisfies this test case and violates the loop assertion. This boundary case focuses attention onto Low, where it is determined that Low should be initialized to 0.

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

4.4 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 at line 15, generate the following path representation:

PC	[((n/2+1)/2)**2 > n or ((n/2+1)/2+1)**2 ≤ n] and ((n/2+1)/2+1)**2 ≤ 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 generate the following test case:

input	((n/2+1)/2)**2 ≤ n and 0=(n/2+1)/2 and (n > (n/2+1)/2)
accept	*false*

The value n=1 satisfies the input condition. This highlights the inconsistency between the annotation Low ≤ Mid and the assignment Low := Low+1.

4.5 Oracle/Fault-Based Testing

Suppose we hypothesize that

High := N/2+1

should be

High := (N+1)//2

The revealing condition is:

origination condition	N/2+1 ≠ (N+1)/2
transfer condition	trivial (true)

In order to violate the negated loop assertion, Low**2 > N or N > High**2, we choose N = 2. This 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.

5 Specification-Based Testing with Larch

5.1 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.

5.1.1 Larch Shared Language

Larch's shared language extends the capabilities of algebraic specification. The extensions do not increase the power of the specification language, 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 an 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.[6]

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 specifies that the results have been left unspecified intentionally.

[6] 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.

```
STACKOFINT: trait
    imports INTEGER
    introduces
        EMPTY: → INTSTACK
        PUSH: INTSTACK x INTEGER → INTSTACK                                    5
        TOP: INTSTACK → INTEGER
        REST: INTSTACK → INTSTACK
        SIZE: INTSTACK → INTEGER
    closes INTSTACK over [NEW,PUSH]
    partitioned by [TOP,REST]                                                  10
    constrains [INTSTACK] for all [S:INTSTACK;I:INTEGER]
        REST(PUSH(S,I)) = S
        REST(EMPTY) EXEMPT
        TOP(PUSH(S,I)) = I
        TOP(EMPTY) = ERROR                                                     15
        SIZE(EMPTY) = 0
        SIZE(PUSH(S,I)) = 1 + SIZE(S)
```

Figure 5: Larch shared language specification of a stack of integers

5.1.2 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 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 designed an Ada interface language.[7] Figure 6 provides an interface specification 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 to it. 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 specifications can be incomplete or non-deterministic. A **mutates** clause in the middle of a pre/post pair indicates the parameters or

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

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

    function Empty(S:Stack) returns Boolean is
        pre: true                                                    5
        post Empty = (SIZE(S) = 0)
    end;

    function Full(S:Stack) returns Boolean is
        pre: true                                                    10
        post: Full = (SIZE(S) = 100)
    end;

    function Top(S:Stack) returns Integer is
        pre: true                                                    15
        post: Top = TOP(S)
    end;

    function Create return Stack is
        pre: true                                                    20
        post: Create = EMPTY and new(Create)
    end;

    procedure Push (S: in out Stack; E: Integer) is
        pre: SIZE(S) ≤ 100                                           25
        mutates: S
        post: S_post = PUSH(S_pre,E)
    end;

    procedure Pop (S: in out Stack; I: out Integer) is               30
        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                                       35
    end;
end;
```

Figure 6: The interface language specification of a stack

```
package IStack 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)                                          40
    end Rest;
end IStack;
```

Figure 7: Mapping Function Between the Concrete and Abstract Values

global variables that can be modified by the function. If no mutates clause is present, then no variables can be modified. 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.

5.1.3 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 is useful. 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].

One way to define the mapping between the abstract and concrete is to require that the interface portion of the specification includes a mapping from each function in the trait in a **partitioned by** clause to an implemented function. 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 example is extended with the mapping information.

By using the equivalent abstract and concrete functions and recursing through the data structure, it is possible to find the set of conditions for equivalence. In the stack example, 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 is given. If the shared language rules for the functions in the partitioned by

Concrete Value:	(1,2,3)
Conditions:	top(S1) = 1 rest(S1) = S2 top(S2) = 2 rest(S2) = s3 top(S3) = error rest(S3) = error
Solution:	S1 = push(3,push(2,push(1,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                                          5
end SquareRoot;
```

Figure 9: Larch Interface Specification for a Square Root Function

clause are complete, then there is only one solution to the generated 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.

5.2 Spec/Error-Based Testing

Error-based testing techniques could be applied to Larch pre/post-conditions by using boundary value testing. One means of boundary testing is to recursively analyze each expression in the condition in the following way. For each expression, if the operator is *and*, remove each subexpression one at a time and generate test data that violates the removed condition and test data that satisfies it. If it is impossible to violate the removed condition, then the condition was not necessary. On the other hand, if the operator is *or*, then generate test data that will cause each subexpression to be come true, while the others are false. An additional test case where all of the subexpressions is true could also be useful. If any test cases are infeasible, the specification is incomplete and should be analyzed further to check for faults. Several test case descriptions are given below as examples of this method being applied to

the SquareRoot function, from Figure 9.

Input:	(N < 0) and (SquareRoot ** 2 > N) and (N < (SquareRoot+1) ** 2)
Accept:	((N ≥ 0) and (SquareRoot ** 2 ≤ N) and (N < (SquareRoot+1) **2)) or ((N < 0) and (raise Domain_Error))

If the pre-condition is not changed, then the acceptance criteria can be the post-condition, because the same post-condition should still hold.

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)

5.3 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 proposed in either tier. Here we consider a fault in the interface language. The proposed fault is that the stack example in Figure 6 had the following post-condition for Pop:

$$I_{post} = \text{TOP}(\text{REST}(S_{pre})) \text{ and } S_{post} = \text{REST}(S_{pre})$$

The following test case distinguishes between the alternatives.

Input:	$\text{TOP}(\text{REST}(S_{pre})) \neq \text{TOP}(S_{pre})$
Accept:	false

If such a test case can be found, then it will show a difference between the specification and the variant. Furthermore, if the tester checks the input/output pair and accepts it, then the fault is not in the specification. If the output is incorrect, on the other hand, and the alternate produces the correct output, a fault has been found in the specification.

5.4 Oracle/Error-Based Testing

Here we apply oracle/Error-based testing to to the corrected Ada code for SquareRoot given in Figure 4 and the Larch specification in Figure 9. As an example, pick the following path (8,10,12,13,10,12,13,10,20,21) for which the following symbolic representation is generated:

Path Condition:	$(\frac{(N+1)^2}{16} > N)$ and $(\frac{(N+1)^2}{64} > N)$ and $(\frac{(N+1)^2}{256} \leq N)$ and $(\frac{(N+17)^2}{256} > N)$
Path Computation:	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, which tests the points where the specification is incomplete. In Larch, incompleteness is introduced by **exempt** clauses or missing cases. **Exempt** clauses in the shared language specification represent **terms** that do not have a specified value in the trait's theory, while missing cases are input ranges that violate all of the pre-conditions for the function. In 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*

5.5 Oracle/Fault-Based Testing

The RELAY model can be used to find the revealing conditions for a hypothesized fault of adding a constant value to an expression in Push[8] such as:

[8] Figure 3, line 32.

Original:	Stack.Elems(Stack.Count) := Item;
Hypothesized:	Stack.Elems(Stack.Count + k) := Item;

In order to cause a differentiation, one of the implementations must satisfy the specification, and the other must not. The revealing condition is:

$$(variant(\text{Stack}_{post}) \neq \text{PUSH(Item,Stack}_{pre}\text{) and Stack}_{post} = \text{PUSH(Item,Stack}_{pre}\text{)) or}$$
$$(variant(\text{Stack}_{post}) = \text{PUSH(Item,Stack}_{pre}\text{) and Stack}_{post} \neq \text{PUSH(Item,Stack}_{pre}\text{))}$$

Because of the **partitioned** clause in the specification in Figure 5, each 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:

$$(\text{TOP(variant(Stack}_{post}\text{)) } \neq \text{ Item and TOP(Stack}_{post}\text{) = Item) or}$$
$$(\text{TOP(variant(Stack}_{post}\text{)) = Item and TOP(Stack}_{post}\text{) } \neq \text{ Item)}$$

Symbolic evaluation of the original and variant Push, followed by Top, generates the following conditions:

Path Condition:	stack.count $\leq$ Elem_List'LAST
Path Value:	Result = (fetch stack.count+1 (store stack.count+1 item stack.elems))
Variant Path Value:	Result = (fetch stack.count+1 (store stack.count+1+k item stack.elems))

By evaluating the previous condition with these symbolic values, the following test case description is generated.

Input:	((fetch stack.count+1 item stack.elems) $\neq$ item) and (stack.count $\leq$ Elem_List'Last)
Accept:	*ok*

If each element of stack.elems_{pre} has a different value than item, the stack is not full, and the test case executes correctly, the hypothesized fault is not present.

6 Conclusion

In this paper, we describe our research in extending the ideas underlying implementation-based testing techniques to generate test cases 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. Moreover, we 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 must formalize these specification-based approaches for test case generation. We must 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 an environment that contains generic testing and analysis capabilities designed to be language independent. We are currently writing a front-end for Anna, which will translate into TEAM's common internal form, and developing symbolic evaluation capabilities for Anna. This will enable us to use the test case generation capabilities in TEAM. We intend to do the same for other specification languages.

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. 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.

[CPRZ85] Lori A. Clarke, Andy Podgurski, Debra J. Richardson, and Steven J. Zeil. "A Comparison of Data Flow Path Selection Criteria". In *Proceedings of the Eighth International Conference on Software Engineering*, pages 244–251, London, England, August 1985.

[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, Boston, MA, November 1988.

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

[GG75] J.B. Goodenough and S.L. Gerhart. "Toward a Theory of Test Data Selection". *IEEE Transactions on Software Engineering*, SE-1(2):156–173, June 1975.

[GHM87] John Gannon, Richard Hamlet, and Harlan Mills. "Theory of Modules". *IEEE Transactions on Software Engineering*, SE-13:820–829, 1987.

[GHW85] John Guttag, James Horning, and Jeannette Wing. "The Larch Family of Specification Languages". *IEEE Transactions on Software Engineering*, pages 24–36, September 1985.

[GM] M.-C. Gaudel and B. Marre. "Algebraic Specifications and Software Testing: Theory and Application". In *Rapport LRI #407*.

[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.

[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*, SE-11(1), January 1985.

[Las88] Janusz Laski. "Testing in Top-Down Program Development". In *Second Workshop on Software Testing, Verification, and Analysis*, pages 72–79, July 1988.

[LvH85] David C. Luckham and Friedrich W. von Henke. "An Overview of Anna, a Specification Langage for Ada". *IEEE Software*, pages 9–22, March 1985.

[LvHKBO84] David C. Luckham, Friedrich W. von Henke, Bernd Krieg-Brueckner, and Olaf Owe. "ANNA: A Language for Annotating Ada Programs". Technical Report 84-261, Stanford University, July 1984.

[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 J. Ostrand, Ron Sigal, and Elaine J. Weyuker. "Design for a Tool to Manage Specification-Based Testing". *IEEE Transactions on Software Engineering*, SE-12:41–50, 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.

[RT86] Debra J. Richardson and Margaret C. Thompson. "RELAY: A New Model of Error Detection". Technical Report 86-64, Computer and Information Science, University of Massachusetts, Amherst, December 1986.

[RT88] Debra Richardson and Margaret Thompson. "The RELAY Model of Error Detection". In *Second Workshop on Software Testing, Verification and Analysis.* IEEE CS Software Engineering Technical Committee, July 1988.

[Vel87] F.R.D. Velasco. "A Method for Test Data Selection". *The Journal of Systems and Software*, 7:89–97, 1987.

[WC80] L.J. White and E.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, Massachussetts Institute of Technology, 1983.

[WO80] E.J. Weyuker and T.J. Ostrand. "Theories of Program Testing and the Application of Revealing Subdomains". *IEEE Transactions on Software Engineering*, SE-6(3):236–246, May 1980.