DESIGNING FOR SOFTWARE TESTABILITY USING AUTOMATED ORACLES*

James M. Bieman Hwei Yin

Department of Computer Science
Colorado State University
Fort Collins, Colorado 80523 USA
(303) 491-7096
bieman@cs.colostate.edu, yin@cs.colostate.edu

Published in *Proc. International Test Conference*, pp. 900–907, September 1992 (ITC92).

ABSTRACT

Software testing often requires massive numbers of test cases that must be manually inspected for correctness. This paper demonstrates the use of software "test oracles" to automate the process of checking the correctness of program output. The Prosper system, implemented by the authors, can be used to define test oracles and monitor the runtime behavior of software. An effective method to design software for testability must include the concurrent development of test oracles.

1 INTRODUCTION

Both path-based (white box) software testing and the random statistical (black box) software testing used in software reliability modeling usually require the execution of massive numbers of test cases. The testing process must be automated to effectively complete such massive testing. Automated testing requires (1) an automated means for generating the test cases, (2) a test harness to execute the software of interest on the test data, and (3) automated *test oracles* — automatic means for determining if program output is correct.

Our focus is on the test oracle problem. Unfortunately, in most large-scale software development environments, human labor is the primary means employed to monitor the correctness of test case execution. Engineers may either manually derive correct output values for each test case generated, or may manually inspect the output after each test run. This labor intensive process is both expensive and error prone. The process of determining the correctness of program output must be automated to support the amount of testing required to develop measurably reliable software systems.

Automated test oracles can be based on some form of executable specifications. In this paper, we demonstrate a test oracle design technique and system (Prosper) that we have developed. We use the Prosper executable specification language to specify test oracles. Using a run time type system, we can automatically monitor a program to determine if its execution is consistent with its Prosper test oracle. Run time software errors are flagged by the type system.

To be effective on large software systems, executable test oracles must be developed along with the software design and implementation. The software design process must include an oracle design component. We aim to develop software that can be tested automatically using associated oracles.

2 TEST ORACLES IN PROSPER

The Prosper type system can ensure that program post-conditions are satisfied when post-conditions are written as Prosper Boolean functions.

Consider a program P that inputs data of type $T1$ producing output of type $T2$. P has a type signature:

$$P\text{: } T1 \rightarrow T2.$$

A post-condition *P-post* specifying the correctness of the output from P can be written as a Prosper Boolean

*This research was partially supported by the Colorado Advanced Software Institute (CASI) and Storage Technology Corp. through a Technology Transfer Grant, "Effective Specifications for Software Reliability Management."

function with the type signature:

P-post: $T1 \times T2 \rightarrow bool$.

P-post takes an input value and an output value and returns true if the output value satisfies the specification for P.

We can also define *P-post* in a *Curried* fashion; a Curried function processes its arguments one at a time:

P-post': $T1 \rightarrow (T2 \rightarrow bool)$.

The Curried *P-post'* inputs an argument of type $T1$, and outputs a function of type $T2 \rightarrow bool$. When given an argument of type $T1$, *P-post'* partially evaluates, creating a function of type $T2 \rightarrow bool$. Later, when the output of P is known, it can be processed by the partially evaluated *P-post'* to produce the Boolean result.

The Prosper *select* construct is a built-in function that defines a new type based on a characteristic function. It can use functions like the Curried *P-post'* to define membership in a user defined type. Using *select* we can strengthen the type signature of P to

P(x): T1$\rightarrow$ (*select(P-post'(x))*)

In the above type signature, parameter x binds to the input argument of P and is used as an argument to the *P-post'* function. After P receives an input value, the output type is dynamically created to allow only those outputs that are correct relative to x. *P-post'(x)* is used by the Prosper *select* construct to automatically build an output checker for program P. When P is activated with input x, *P-post'(x)* is invoked creating a characteristic function that describes a type relative to x. Thus, *P-post'* partially executes and waits for $P(x)$ to produce its result. The output of $P(x)$ then becomes the input to the partially executed *P-post'*. This output checking is equivalent to running *P-post'* as follows:

(P-post'(x))(P(x)).

Prosper automatically performs this checking and incorrect output is flagged. If program P produces incorrect results, the Prosper system will identify the error as a type violation. The execution of massive numbers of test cases is possible when Prosper-like test oracles are designed as part of a software system.

A unique aspect of Prosper is a mixing between objects from the "world" of types and objects from the "world" of values. Type expressions can accept values as parameters, and functions can accept or return types as if they are values. In the forgoing example, x is the input parameter of P; x is also the input to the *P-post'* function, which is used by the *select* function to define a new type. This kind of computation is unusual, and can be used to design test oracles.

2.1 The Prosper System

Our system uses a Lisp-like syntax, is implemented in Lisp, and executes on a Sun SPARC workstation [24]. Prosper (PROtotypes and SPEcifications with Relative types) was originally defined by Leszczyłowski and Bieman [15]. Our implementation includes enhancements to several aspects of the original definition. One improvement is the inclusion of Prosper *select* expressions, a mechanism for defining types in terms of characteristic functions.

Prosper computation is based on expression evaluation; therefore, expressions must be used to describe Prosper specifications. When an expression is fully evaluated, both the expression value and type are returned.

Prosper represents all of its values as *value cells*. Value cells consist of two parts: a value and its type (or *domain*). These parts are separated with an "`@`" symbol, as in `(3 @ integer)` or `(true @ boolean)`. Type checking ensures that the type part of the cell is the appropriate higher world classification of the value part. Thus, the value cell `("ab" @ string)` asserts that the quantity `"ab"` is part of the domain `string`. When value cells are entered explicitly, a type check is performed to ensure that the quantity and domain entered are consistent. The interpreter evaluates a value cell to itself. User input follows the "$>$" character in the following examples of value cell evaluation:

```
> (3 @ integer)
(3 @ INTEGER)

> ("true" @ boolean)
(*ERROR* ACTUAL-STRING, EXPECTED-BOOLEAN)

> ((3 @ integer) @ integer)
(3 @ INTEGER)
```

Note that we have made a few minor syntax simplifications to aid the reader in the Prosper code shown in this paper.

Prosper uses its type system to enforce invariants, expressed as Boolean functions, at run time. Consider a Prosper type, `intlist`, used to represent integer lists. We can write assertions concerning integer lists as Prosper Boolean functions. For example, an assertion that an `intlist` value is sorted can be defined as a Boolean function *IsOrdered* with the following type expression:

IsOrdered: `intlist`$\rightarrow$*Boolean*

In Prosper, the *IsOrdered* function is defined by the following Lisp-like routine:

```
(define IsOrdered
 (fun L:intlist boolean
  (cond ((< (length L) 2) true)
        ((<= (first L) (first (tail L)))
         (IsOrdered (tail L)))
        (true false))))
```

Function `IsOrdered` inputs an `intlist` value which is bound to the parameter `L` and outputs a Boolean value. The body of the function is a straightforward comparison of successive elements.

If we want an `intlist` variable to be maintained as an ordered list, we can define a new Prosper type `SortedList` which uses `IsOrdered` as a characteristic function:

```
(define SortedList (select IsOrdered))
```

The Prosper `select` function generates a new type from the characteristic function. In the example, `select` uses `IsOrdered` as a characteristic function to define the type `SortedList`. Whenever an `intlist` value is bound to a parameter or identifier specified as of type `SortedList`, the Prosper type checker will determine whether the list is ordered using the `IsOrdered` function. An attempt to bind an unordered list to a `SortedList` parameter will cause a run time type error.

Invariants are specified as (possibly) higher order Boolean functions, and the monitoring of invariants is performed by the type checking system at run time. We can use invariant monitoring for testing oracle purposes.

2.2 A Sorting Program Oracle

The output of a function `Sort` that sorts `intlist` values must satisfy the following post condition (expressed using Prosper Boolean functions):

`IsOrdered(Sort(L))` $\wedge$ `Permutation(L,Sort(L))`

which states that the output of `Sort` is an ordered permutation of its input.

We can use a *curried* Prosper function `SortPost` which processes arguments one at a time:

`SortPost: intlist` $\rightarrow$ (`intlist` $\rightarrow$ `boolean`)

The first input to the curried function is an `intlist`, say `L`, and the output is a new function. This new function takes a second `intlist` as input and returns `true` only when the second `intlist` is a sorted permutation of the first `intlist L`.

The Prosper test oracle function `SortOracle` monitors the correctness of the enclosed sorting program, `Sort` (text following the semicolon are comments.):

```
(define SortOracle
 (fun L:intlist              ; input type
      (select (SortPost L)) ; output check
      (Sort L)))            ; function body
```

where `Sort` is a sorting function with type signature

`Sort: intlist` $\rightarrow$ `intlist`

Correctness monitoring is specified in the second and third lines of the above function. We specify that the input to `SortOracle` is an `intlist` object `L`. We also specify in line 3, `(select (SortPost L))`, that the output type must satisfy the characteristic function produced by `SortPost` with input list `L`. That is, `(SortPost L)` produces a new function that evaluates to true only when the list produced by `Sort` is a permutation of `L` and ordered. The output of `Sort` must satisfy this characteristic function when monitored by `SortOracle`. `SortPost` may be of the following form:

```
(define SortPost
 (fun L1:intlist            ; accepts list L1
      (intlist -> boolean) ; returns a function

 ; SortPost produces the following function:
 (fun L2:intlist boolean
      (and (IsOrdered L2)
           (Permutation L1 L2)))
```

Figure 1 shows the monitoring process of the sorting program. The execution of `SortOracle` and monitoring of `Sort` takes place as follows:

1. Assume `SortOracle` is invoked with list `M` as input: `(SortOracle M)`
2. The input list `M` is bound to formal parameter `L`.
3. The sorting program, `Sort`, is invoked with list `M`. `Sort` produces a new (hopefully) sorted list as output.
4. `SortPost` is invoked with list `M` as input. `SortPost` produces a characteristic function as output. This characteristic function accepts a list as input and outputs true if and only if the input is ordered and is a permutation of `M`.
5. `select` uses the characteristic function generated in 4 above to define a type based on the specification defined by `SortPost`.
6. The output of `Sort` is checked to see if it is of the type defined in 5 above.

Thus, incorrect output is flagged at run time. We invoke `SortOracle` with input list `(7 3 5)`:

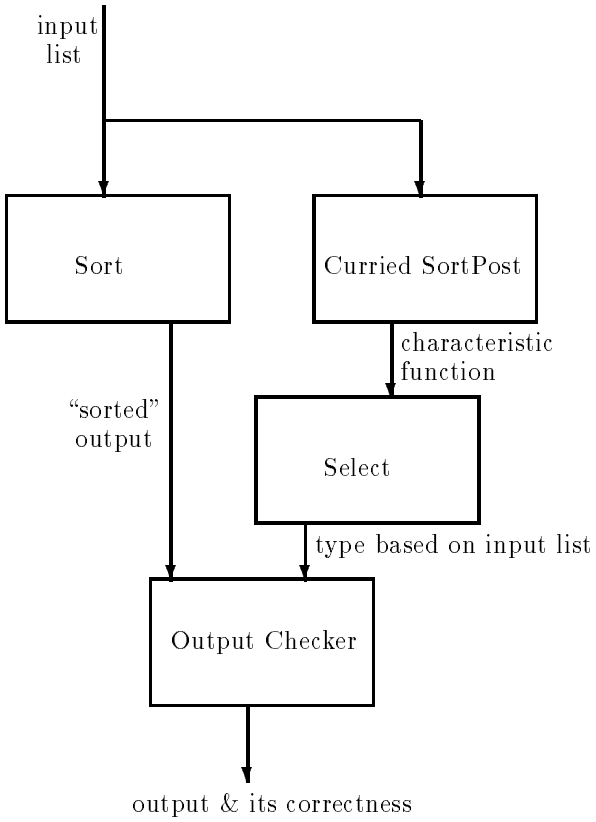

Figure 1: SortOracle data flow

```
> (SortOracle ((7 3 5)@intlist))
((3 5 7)@intlist)
```

The correctness monitoring specified by line two and three of `SortOracle` is the test oracle for the `Sort`. `Sort` is the actual sorting program that is to be tested. Many possible sorting programs can be used. One example is a selection sort:

```
(define Sort
 (fun L:intlist intlist
   (cond
    ;an empty list is sorted
    ((empty L) L

    ;Remove smallest element, recurse on tail.
    ;Put smallest element back in front of list.
    (true (define FirstEl (smallest L))
      (cons FirstEl
            (sort (delete FirstEl L)))))))))
```

where `delete` produces a list without the first occurrence of `FirstEl` in `L`.

Notice that the type specification for `Sort` in line 2 specifies the output as an `intlist` without any stronger restriction. All testing oracle work is performed by `SortOracle`, the caller of `Sort`. Although we have separated the test oracle from the sorting algorithm implementation, we can still perform correctness monitoring by calling `Sort` from within `SortOracle`. This separation is not necessary, but it allows an engineer to strip away the oracle to improve performance after testing.

Revealing a Defect.

A common error in student Lisp programs is to skip the recursive call in the expression that produces the result. This error causes the last two lines in `Sort` to be written without the recursive call:

```
(cons FirstEl (delete FirstEl L))
```

rather than the correct version:

```
(cons FirstEl (sort (delete FirstEl L)))
```

We now run the defective sorting program on the sample list `(7 3 5)`:

```
> (Sort ((7 3 5)@intlist))
((3 7 5)@intlist)
```

Only the lowest element in the input list is in the correct position. When we use the `SortOracle` to monitor the process, the incorrect output is discovered by the output type checker:

```
> (SortOracle ((7 3 5)@intlist))
(*ERROR* NOT OF SELECT TYPE)
```

`SortOracle` will give an error message whenever `Sort` produces an output list that does not satisfy the specification.

The error that we introduced produces incorrect output for the majority of input lists. The output is correct only when the input list is already sorted, or when the smallest element is the only element not in the correct sorted position. Program defects are much more difficult to detect when they are exposed by only a small percentage of possible input values.

Monitoring Random Tests.

We demonstrate how defects that cause infrequent errors are monitored. Suppose that the delete routine that is used by `Sort`, "`(delete n L)`", deletes *all* occurrences of integer `n` in list `L` rather than just the first occurrence. If this version of `delete` is used to implement `Sort`, `Sort` will fail only on input lists with duplicate elements. Output on these lists will be sorted,

but they will not be permutations of the input because they will be missing the duplicates. In all other cases, `Sort` will produce correct results.

We tested `SortOracle` using this defective version of `Sort`. We ran `SortOracle` on 1000 input lists of five elements, with each element selected randomly from the integers between 1 and 1000. The probability that a test case contains a duplicate is .00965. `SortOracle` found defective output in 10 of the 1000 test cases that we ran. Thus, the number of test runs that failed to satisfy our `SortOracle` is reasonably close to the expected failure rate.

In operational use, a testing oracle is run with programs and data containing unknown defects. Random tests can find defects as long as the number of test cases is large enough. This is to ensure a high probability of finding data that causes the incorrect output. The ratio of invalid tests detected by the oracle to the total number of cases is a measure of the reliability of the tested software. Thus, an automated oracle can be used as a software reliability measurement tool.

It is possible to manually scan program output searching for errors. However, when large numbers of test cases are run, manual correctness checking is impractical and error prone. Scanning the output from our 1000 test cases of the defective `Sort` program is a grueling task. It is easy to miss the incorrect output, especially if the errors are rare. The `SortOracle`, however, detects all such errors automatically and accurately.

3 DESIGNING WITH ORACLES

Many defects are much rarer than the one in the defective `Sort` program described above. Spotting defective output that occurs infrequently is nearly impossible when defect detection relies solely on human analysis. Therefore, automated testing oracles are essential for detecting infrequent errors in a software product. Testing oracles are a form of parallel code that checks rather than produces output.

Oracle development can represent significant effort which may increase design and implementation cost; however, overall testing and maintenance costs should be reduced. Commercial software systems are many orders of magnitude more complex than the example sorting program; they may have thousands of interconnected modules. Oracle development must therefore be carefully integrated into the software development life cycle. Oracles must be designed for unit testing, subsystem testing, and integration testing in a disciplined manner.

Oracles can be designed directly from formal design specifications [21]. The purpose of a formal specification is to describe the behavior of a software system at the highest level of abstraction without including implementation details. A specification that is executable can be tested to determine if its behavior satisfies informal requirements. The term "executable specification" usually refers to specifications that produce rather than check output. There are a number of approaches to developing formal specification [8, 13, 6]. Most techniques are based on either algebra or set mathematics [6, 10]. These techniques are not, in general, executable due to the abstract nature of some of the constructs used. Some researchers claim that specifications should not execute so that implementation details can be left out [9]. Despite these criticisms, many specification languages including Paisley, OBJ, me too, RSF, and Prosper are executable [25, 7, 11, 5, 15, 1]. Another use for specification is to annotate a design or implementation. Anna [17] annotates Ada programs, and A++ annotates C++ programs [4]. Anna is partially executable. A Prosper test oracle is also a form of an executable specification. It must execute in order to check the correctness of output. However, the oracle does not have to *produce* the specified output.

Often, a specification describes the system in terms of mathematical entities rather than programming language data types [12]. When designing an implementation from such a specification, the developer must map implementation objects (programming language data types) to the more abstract objects of the specification. To develop a test oracle, this mapping must be explicit; the developer must specify a mechanism to convert program output into values that the test oracle can process.

Three *name spaces* can be used when designing with test oracles: a specification name space, an oracle name space, and an implementation name space [21]. The specification name space is based on abstract mathematical objects. The implementation name space is based on data manipulated by program code. The oracle name space lies somewhere in between. Mappings must be defined to show how objects in one name space relate to objects in another. Values in the implementation name space must be converted to values in the oracle name space before the oracle can execute.

We propose that software be designed using the following process:

1. Develop an abstract specification that includes post conditions for the output of operations. The specification and associated post conditions can be expressed in natural language. However, a formal specification increases precision at an early stage.

2. Determine what implementation data structures will represent objects in the abstract specification.

3. Design (and implement) oracles that can determine whether the output from program operations satisfy the specifications. Oracles can be designed to focus on the most critical component of the specification.

4. Design and implement program functions using the specification from Step 1.

5. Design a mechanism to automatically generate test data. Since the oracle function is automated, large sets of test data can be used. The data sets can be tailored to specific structural testing criteria.

6. Run and monitor the implemented functions using the test data.

7. Correct the program (or test oracle) and re-run the tests

To maximize the reliability of a system, we suggest that the above steps be performed for all program units and subsystems. Since much of the testing process is automated, testing can begin as soon as the program units and their associated oracles are complete. Testing can be performed concurrently with development.

An automated (or semi-automated) mechanism for generating large sets of test data can be used when designing with test oracles. A random test case generator, for example, may be used to generate any size set of test data. If the random data is consistent with the expected use of the software (following the operational profile), then test results can be used to measure the current reliability of the software. Random testing can be used along with structural test coverage criteria [18, 23, 22, 3, 20, 19]. Random data can be run until it is determined that the desired criteria are satisfied. Alternatively, data can be generated to specifically satisfy particular testing criteria [2]. The use of test oracles does not depend on using random testing or structural testing. However, the effort of developing test oracles is justified when a large number of test runs are required.

Test oracles can be set up to run on-line so that output can be checked as it is produced. Our Prosper system is currently designed to perform oracle functions on-line. Another arrangement is to dump program output (and other necessary state information) to a file so that the oracle can check program correctness as a separate process. For each function being tested, an oracle system needs the following key information:

- input state.
- output state.
- timing information (especially for real time systems).

The oracle compares the input state to the output state and checks for consistency with the specification. The exact architecture of the oracle system must address the overhead of running the test oracles.

Oracles can improve software correctness and reliability. However, oracles can also reduce performance. The effects on performance are most serious for real time software. Oracle execution requires processor time and memory which can seriously reduce performance. One strategy is to use the oracles during testing, and then remove them when the software is released. However, removing oracles can cause unexpected problems. Test oracles are essentially software probes, and removing probes may change the timing behavior of the system. Timing attributes can be critical for real time systems. Oracle software can be run independently from a real time system to avoid changes in timing behavior.

Since the test oracle is a program it can also contain defects, An error detected by the test oracle may actually be the result of a defective oracle rather than defective code. The source of a defect, whether it is a program defect or an oracle defect, must be found during the debugging process. Thus testing checks both the oracle and the program, and insures that they are consistent.

It is possible for incorrect output to be accepted by an incorrect oracle — both the oracle and program can fail on the same data. Such *coincidental correctness* is a serious problem recognized in studies of *n-version programming* [14, 16]. In n-version programming several implementations are developed for the same specification. The execution behavior of the different versions are compared. Coincidental correctness between a program and its test oracle is unlikely because of different objectives: a program must produce an answer, while an oracle simply checks that answer. However, coincidental correctness is still possible. If an incorrect oracle is used as a specification then the errors may be propagated to the implementation. Such errors will not be discovered during oracle-based testing. Also note that oracles and programs should not share subroutines. Consider the `Sort` program in Section 2.2 with the defective `delete` subroutine. The error would not be detected if the oracle code in `SortPost` used the same `delete` subroutine.

A common problem in software maintenance occurs when program code is modified but the documentation is not. Over time, engineers learn not to trust documentation and, as a result, only the program code can

be used to help understand the system. Test oracles are actually a formalized form of documentation. Consistency between test oracle documentation and the program implementation can be insured. Programs can be tested using the oracles after modifications. The oracles will be updated as required to complete this regression testing process. With such an arrangement, test oracles can provide another accurate description of a software system.

4 CONCLUSIONS

Automated test oracles are critical to any automated software testing process. We use the Prosper system as a test oracle mechanism to monitor functional programs. In Prosper, correctness properties are defined and enforced using a run time type system. We show that Prosper can be effective in finding defects that are revealed in less than 1% of randomly generated test data. If we were to rely on human inspections of program output, such errors would likely be missed.

Without test oracles, rigorous testing is not possible. Oracles allow the testing of large amounts of data. Therefore, they are appropriate for use with the most discriminating testing criteria, random testing techniques, and reliability measurement. Effective test oracles must be designed before (or concurrently with) the implementation of software. Oracle development should be part of the software development process.

We continue to experiment with the Prosper system. We are also developing an oracle notation for programs implemented in C.

REFERENCES

[1] J.M. Bieman and H. Yin. Monitoring the correctness of software. *Proc. ISMM Int. Symp. Engineering & Industrial Applications*, pp. 79–82, Dec. 1991.

[2] L. A. Clarke. A system to generate test data and symbolically execute programs. *IEEE Trans. Software Engineering*, SE-2(5):215–222, Sept. 1976.

[3] L. A. Clarke, A. Podgurski, D. J. Richardson, and S. J. Zeil. A comparison of data flow path selection criteria. *Proc. 8th Int. Conf. Software Engineering*, pp. 244–251, 1985.

[4] M. P. Cline and D. Lea. The behavior of C++ classes. *Proc. Symp. OOP Practical Application*, pp. 81–91, 1990.

[5] M. Degl'Innocenti, G. L. Ferrari, G. Pacini, and F. Turini. RSF: a formalism for executable requirements specifications. *IEEE Trans. Software Engineering*, 16(11):1235–1246, Nov. 1990.

[6] S.J. Garland, J.V. Guttag, and J.J. Horning. Debugging larch shared language specifications. *IEEE Trans. Software Engineering*, 16(9):1044–1057, Sept. 1990.

[7] J. Goguen and J. Meseguer. Rapid prototyping in the OBJ executable specification language. *ACM Sigsoft Software Engineering Notes*, 7(5), 1982.

[8] A. Hall. Seven myths of formal methods. *IEEE Software*, 7(5):11–19, Sept. 1990.

[9] I.J. Hayes and C.B. Jones. Specifications are not (necessarily) executable. *Software Engineering Journal*, 4(6):330–338, Nov. 1989.

[10] I. Hayes (editor). *Specification Case Studies*. Prentice-Hall, London, 1987.

[11] P. Henderson, C. Minkowitz, and J. S. Rowles. me too *Reference Manual*. STC Technology Ltd., Staffordshire, 1987.

[12] C.B. Jones. *Software Development: A Rigorous Approach*. Prentice-Hall, London, 1980.

[13] C.B. Jones. *Systematic Software Development Using VDM*. Prentice-Hall, London, 1986.

[14] J. Knight and N. Leveson. An experimental evaluation of the assumption of indepaendence in multiversion programming. *IEEE Trans. Software Engineering*, SE-12(1):96–109, Jan. 1986.

[15] J. Leszczyłowski and J.M. Bieman. PROSPER: A language for specification by prototyping. *Computer Languages*, 14(3):165–180, 1989.

[16] B. Littlewood and D. R. Miller. Conceptual modeling of coincident failures in multiversion software. *IEEE Trans. Software Engineering*, 15(12):1596–1614, Dec. 1989.

[17] Donald C. Luckham and Friedrich W. von Henke. An overview of ANNA, a specification language for ADA. *IEEE Software*, pp. 9–22, March 1985.

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

[19] S. C. Ntafos. A comparison of some structural testing strategies. *IEEE Trans. Software Engineering*, 14:868–874, June 1988.

[20] S. Rapps and E. J. Weyuker. Selecting software test data using data flow information. *IEEE Trans. Software Engineering*, SE-11(4):367–375, April 1985.

[21] D. J. Richardson, S. L. Aha, and T. O. O'Malley. Specification-based test oracles for reactive systems. *Proc. 14th Int. Conf. Software Engineering (ICSE-14)*, May 1992 (in press).

[22] M. D. Weiser, J. D. Gannon, and P. R. McMullin. Comparison of structured test coverage metrics. *IEEE Software*, 2(2):80–85, March 1985.

[23] E. J. Weyuker. The complexity of data flow criteria for test data selection. *Information Processing Letters*, 19:103–109, Aug. 1984.

[24] Hwei Yin. Automatic enforcement of invariants: The implementation of Prosper. Master's thesis, Department of Computer Science, Colorado State University, 1991.

[25] P. Zave and W. Schell. Salient features of an executable specification language and its environment. *IEEE Trans. Software Engineering*, SE-12(2):312–325, February 1986.