Discovering Security and Safety Flaws using Property-Based Testing

by

GEORGE CLIVE FINK

B.A. (University of Wisconsin, Madison) 1988
M.S. (University of California, Davis) 1992

DISSERTATION
Submitted in partial satisfaction of the requirements for the degree of
DOCTOR OF PHILOSOPHY

in

Computer Science

in the

OFFICE OF GRADUATE STUDIES
of the
UNIVERSITY OF CALIFORNIA
DAVIS

Approved:

[illegible]

Chair

[illegible]

[illegible]

Committee in Charge

1996

Discovering Security and Safety Flaws using Property-Based Testing

This research was supported by DARPA grant USNN00014-94-1-0065.

Abstract

Property-based testing is a new testing methodology that focuses testing and analysis on the most important properties of a program, making validation results from testing approach the validation results attainable from verification.

Many different errors in programs result from similar generic flaws. Property-based testing provides a practical method for specifying generic flaws and validating that a program does not possess these flaws by testing the program with respect to specifications of these flaws.

Specifications of generic properties in the TASPEC language drive the testing process and control how the other techniques are applied. Test oracles that judge the correctness of program executions are generated automatically from property specifications. Static analysis and program slicing are the most important techniques, as they reduce the problem size for other aspects of the analysis. Iterative contexts, a new definition-use test coverage metric, provides a completeness measure that is stronger than similar definition-use coverage metrics.

Unlike previous testing techniques that provided results exclusively with respect to specifications or code, but not both, property-based testing ties testing results directly to a specified property. Results from the testing process are in the form of coverage with respect to a property.

The Tester's Assistant is an ongoing implementation of property-based testing. As more aspects of property-based testing are implemented and automated, further empirical evaluation of property-based testing will be possible.

Contents

List of Figures **vii**

1 Introduction **1**
- 1.1 The goal of testing . . . 2
- 1.2 Techniques of property-based testing . . . 4
- 1.3 Main ideas . . . 7
- 1.4 Organization . . . 9

2 Property-based Testing Session **10**
- 2.1 Description of ftpd and its flaw . . . 10
- 2.2 Selecting/identifying a property . . . 11
- 2.3 Static analysis and slicing . . . 12
- 2.4 Program instrumentation . . . 13
- 2.5 Initial test executions . . . 13
- 2.6 Coverage evaluation . . . 14
- 2.7 Additional test cases . . . 15
- 2.8 Correctness evaluation . . . 16

3 Background, Methods, and Examples **17**
- 3.1 Computer security and property-based testing . . . 17
  - 3.1.1 Issues in computer security . . . 18
  - 3.1.2 Example programs with security flaws . . . 19
  - 3.1.3 Approaches for establishing security . . . 20
  - 3.1.4 Using property-based testing for security . . . 22
- 3.2 Specifications . . . 22
  - 3.2.1 Specifications and testing . . . 23
- 3.3 Static analysis . . . 25
- 3.4 Slicing . . . 26
  - 3.4.1 Uses of slicing . . . 28
- 3.5 Coverage analysis . . . 29
- 3.6 Debugging . . . 31

4 TASPEC **32**
4.1 Overview 33
4.2 Basic TASPEC concepts 34
4.2.1 Simple TASPEC example 35
4.3 TASPEC language description 36
4.3.1 State elements 37
4.3.2 Compounding operators 37
4.3.3 Location specifiers 39
4.3.4 Event processing 40
4.4 Evaluation model and implementation issues 41
4.5 Completeness 43
4.6 More examples 46
4.6.1 Authentication and ftpd 46
4.6.2 Race conditions and rdist 49
4.6.3 Array overflow in C 51
4.7 Conclusions 54

5 Static data-flow analysis **57**
5.1 Overview 59
5.1.1 Well-behavedness 61
5.2 Internal program representation 63
5.3 Construction of the data-flow graph 68
5.3.1 Pre-processing 68
5.3.2 Live assignment propagation 70
5.3.3 Performance bounds 75
5.4 Program slicing 78
5.4.1 Slicing results 80

6 Iterative Contexts **81**
6.1 Simple coverage metrics 83
6.2 Iterative context coverage metrics 86
6.2.1 Loop-free paths; the base case 86
6.2.2 Contexts in the presence of loops 89
6.3 Calculating iterative contexts 91
6.3.1 Extension to dual program locations 95
6.4 Comparison with other metrics 96
6.4.1 Use- and live-contexts 97
6.4.2 Required k-tuples 98
6.4.3 Path coverage 99
6.5 Summary 100

7 Execution Monitoring **101**
7.1 Correctness monitoring . . . 102
7.2 Completeness monitoring . . . 106
7.3 Instrumentation . . . 110

8 The Tester's Assistant **112**
8.1 Operating the Tester's Assistant . . . 112
8.1.1 Pre-processing the program . . . 112
8.1.2 Running the static analyzer . . . 113
8.1.3 Getting output from the static analyzer . . . 115
8.1.4 Property specifications in the Tester's Assistant . . . 116
8.2 Internals of the Tester's Assistant . . . 116
8.2.1 Lido overview . . . 117
8.2.2 Tester's Assistant architecture . . . 119
8.2.3 Implementing data-flow analysis in ELI . . . 121
8.3 Future directions for the Tester's Assistant . . . 122

9 Results and Conclusions **124**
9.1 Summary . . . 124
9.2 Perspective . . . 126
9.3 Future work . . . 127
9.4 Conclusion . . . 129

A TASPEC Grammar **131**

B Interactive Session with the Slicer **134**

C Abstract Syntax for C **149**

Bibliography **159**

List of Figures

1.1 Property-based testing and the Tester's Assistant. 6

2.1 Ftpd flaw flowchart. 11
2.2 Four initial test cases for ftpd. 14

3.1 Simple example function before and after slicing. 27

4.1 TASPEC properties used in the test of a C program. 33
4.2 Evaluation model for TASPEC. 43
4.3 Property specification for authentication. 48
4.4 Pseudo-code for the rdist race condition. 50
4.5 Partial specification of the race condition window exploited in rdist. . 51
4.6 Specification of static array bounds checking. 52
4.7 Array bounds checking with calloc. 55

5.1 Three examples of potentially ill-behaved code. 62
5.2 The data-flow graph for the factorial function. 66
5.3 Size of the data-flow graph compared to lines of code and size of the parse tree. 68
5.4 Broad outline of the pre-processing step. 69
5.5 AST for the program fragment `a = b + c; printf(a);`. Shading represents nodes which are also in the data-flow graph. 69
5.6 Propagation of live assignments. 71
5.7 Live assignment propagation across structured control constructs. . . 73
5.8 Example of potential for incorrect flow between function call sites. . . 74
5.9 Live assignment list size. 76
5.10 Empirical results from the Tester's Assistant static analyzer. 78
5.11 Slicing results. 80

6.1 A precursor to the Hamlet example. 84
6.2 The basic Hamlet example. 85
6.3 An extension of the Hamlet example. 85

6.4 A short program and coverage contexts. 88
6.5 A loop where dependences build up across iterations. 90
6.6 The gcd program, an example of a numerical algorithm which is hard to test with the Tester's Assistant. 90
6.7 Iterative context construction algorithm. 93
6.8 Computation of contexts. 94
6.9 Sample program with node labels. 95
6.10 Abridged source of ftpd. 97

7.1 Property specification of the *authenticated* predicate. 103
7.2 Specification of *permissions_granted*. 103
7.3 Partial text of the MINIX login program. 105
7.4 Fragment of source of ftpd. 107
7.5 Data-flow graph for ftpd fragment. 108
7.6 Example of code instrumentation, before and after. 110

8.1 Command line arguments for the Tester's Assistant. 113
8.2 Interactive Tester's Assistant commands. 116
8.3 Sample AST rule. 118
8.4 AST attributes. 119

Chapter 1

Introduction

This dissertation introduces a new framework for testing computer programs: *property-based testing* [18, 17, 16]. Property-based testing is a testing process, driven by a specification of a property, that validates a given program for that property. A validation is an assurance that the property holds during any possible execution of the program. For example, assume that a human tester wants to confirm that a specific program correctly authenticates the users of that program. A property-based test of the program satisfies this goal by specifically testing authentication properties and how these authentication properties are implemented in source code.

This dissertation introduces a methodology for implementing property-based testing. In this methodology, a property-based test of a program uses a static analysis of a program to guide dynamic testing. Dynamic testing involves execution of the program with a variety of input data. A new specification language TASPEC (Tester's Assistant SPECification language) was developed for property-based testing. Property specifications expressed in TASPEC are translated automatically into test oracles that check the correctness of program executions.

Explicitly specified relationships between property specifications and source code allows static analysis to be focused on the specific code relevant to the specified property; this technique is called *slicing* [43, 42, 63, 64]. Because they are processing program slices, and not the whole program, testing and analysis algorithms with higher time bounds become practical to use. For example, path-based test coverage metrics are typically viewed as too expensive for practical use [9]. A new path-based coverage metric, *iterative contexts*, efficiently captures the sliced-based computations in the program.

Testing and validation of large programs is not feasible if major portions of the test process are not automated. This dissertation describes the Tester's Assistant, an implementation that automates some aspects of property-based testing for the C programming language [34]. As a demonstration, property-based testing and the Tester's Assistant are applied to several security-critical programs in Unix.

1.1 The goal of testing

Testing computer programs is a critical part of the software development cycle. A test consists of a set of executions of a given program using different input data for each execution, in order to determine whether or not the program functions correctly. A test has a *negative* result if an error is detected during the test (i.e., the program crashes, a property is violated, etc.). A test has a *positive* result if a series of tests produces no error, and the series of tests is "complete" using some coverage metric. A test has an incomplete result if a series of tests produces no errors but the series does not satisfy the coverage metric. Testing is an intuitive approach; assessment of a program is done through program execution, which is the same way that the program is used. Research into testing methodologies augments this intuitive approach by quantifying the results and giving them formal rigor.

Testing methodologies address some basic questions:

1. What is to be accomplished or established via testing?

2. What test data should be used?

3. When has enough testing been carried out?

4. How is it determined if a test is a success or a failure?

The ideal result of testing, and software development in general, is to produce a program which does correctly what it is intended to do and which does not cause problems with the system in which it operates. Property-based testing makes a specific distinction between these two goals. It is not the scope of this research to investigate methods by which to establish the *functional behavior* of a program, which is whether the program does what it is intended to do. Property-based testing addresses *non-functional behavior*; i.e., that the program does not do "bad" things. A central hypothesis of property-based testing is that undesirable side effects of programs can be specified in a generic, program-independent way, and these property specifications can be used to drive testing. The collection of generic properties can be used to test many different programs.

Making the distinction between functional and non-functional behavior reduces the need for complete descriptions of behavior. A library of pre-defined generic properties expressing desired non-functional behaviors could be used to test a large number of programs. Therefore, subsequent functional specifications would be smaller and more easily written because the requirement for including non-functional behavior in the program specification is lifted. For example, the functional specification for a copy file command could be (informally)

> *"it moves the file named as the first argument to a destination named by the second argument,"*

instead of the (perhaps) more complete version

> *"it moves the file named as the first argument to a destination named by the second argument and doesn't use a temporary file in such a way to cause a race condition and doesn't allow an input argument to overflow an internal stack," etc.*

The additional clauses in the second specification are generic properties expressed in a library of pre-defined specifications.

Once the "properties of interest" are determined, the amount of testing required depends upon the program being tested. Suppose the property of interest involves the security state of the computer system on which the program runs. Under these circumstances, a program which alphabetizes a list of names does not require as thorough an inspection as one that regulates network access to the system, because it does not have any of the characteristics that the generic properties highlight. The program that regulates network access is likely to perform many actions described by generic security property specifications, such as accessing password information and authenticating users. Testing such a program for these generic properties is more complex than testing a program whose functionality is entirely disjoint from these properties.

In the general case, it is impossible to execute a program on all possible data. So, testing must approximate or extrapolate results, leading to an incorrect validation in some cases. However, for a testing process to be valuable, it must validate a program with respect to a property with a high degree of certainty. Property-based testing addresses this conflict with iterative contexts, a new data-flow coverage metric introduced in this dissertation.

Another way of obtaining validation is to verify by formal proof. Unfortunately, the cost and complexity of verification are quite large, so verification is not currently a practical solution. But, even if these problems were solved, verification by proof could not be used alone: verification based upon bad assumptions or a faulty system model can result in an incorrect validation. If an operation has an unanticipated side effect during execution in some situations, it has absolutely no chance of being found by verification. While testing has similar problems (bad properties can result in false validations), a test of a program is an actual execution in a real operating environment, so the exact behavior of the program for the given input data can be observed. Unanticipated side effects can be discovered through thorough testing.

It is important to understand the relationship of testing and of formal verification so that the two can be compared. It is the purpose of property-based testing to establish formal validation results through testing.

1.2 Techniques of property-based testing

Validating a property for a program means that whenever the program is executed the property holds. Two important assumptions are made when using property-based testing to obtain this validation. First, the property identified is assumed to capture everything of interest in the program; it is the only thing examined during the tests. It is also possible to test for sets of properties. Second, it is assumed that completeness of testing can be measured structurally, i.e., entirely through examination of the source code and its structure. After the property is identified, the remainder of property-based testing can be classified as structural testing.

In property-based testing, a property is validated by using the property specification to guide dynamic testing of the program. Information derived from the specification determines if a test execution is correct, and also determines how many test executions are needed.

Property specifications have location identifiers that relate property specifications to code structures. The target property is identified, and the program is statically analyzed to derive dependence information. Then code related to the target property and all other code related to that code are targeted for testing. The process of finding a subset of the program which has identical behavior to the full program with respect to the property is called program slicing [43, 42, 63, 64].

The property specification is also translated automatically into a test oracle for the given program. A test oracle checks the validity of a single execution of the program by analyzing the intermediate and final states of the program for inconsistencies with property specifications.

A test suite is a set of test cases for a program. The completeness of a test suite is measured by a new code coverage metric, *iterative contexts*, which is stronger (i.e., detects more flaws) than previously-proposed practical code coverage metrics [39, 49, 9]. An iterative context is a sequence of assignments defining a sub-path of a possible program execution. The assignments are taken from the program slice and represent a possible computation of a value important to the target property. Taken together, all of the contexts represent many of the possible computations of values relevant to the property. It is not possible to represent with a finite set of input data the infinite number of possible computations for some loops, so in those cases iterative contexts will not completely cover all behavior relevant to a property. In a complete test suite, every context must be represented by at least one test execution in the suite.

Therefore, in property-based testing, checking the correctness of each execution

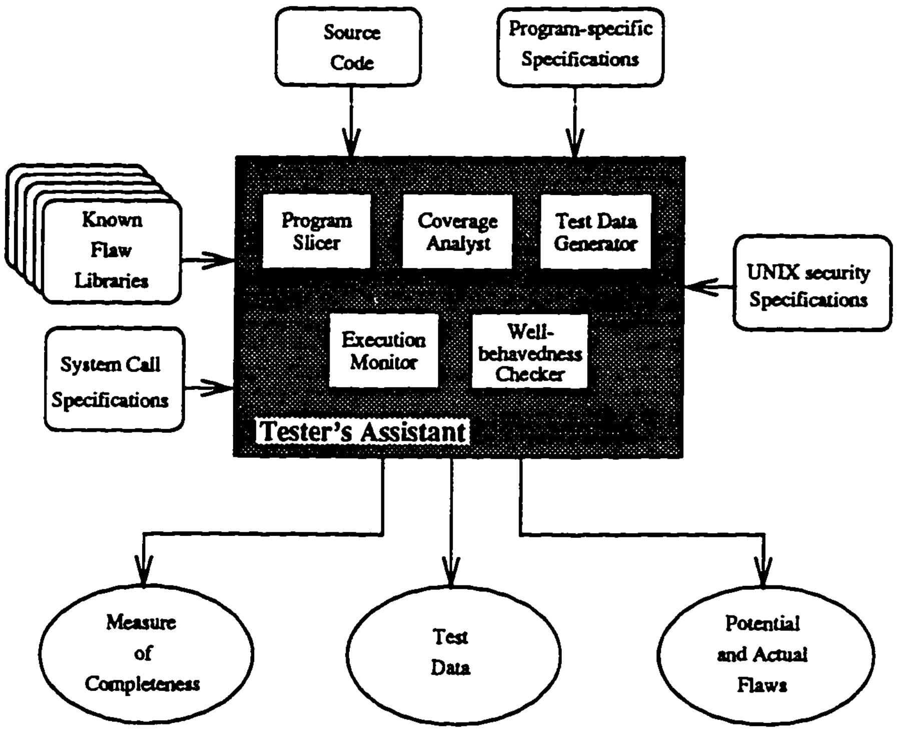


Figure 1.1: Property-based testing and the Tester's Assistant.

together with a description of all the relevant executions of the program establishes the validation of a program with respect to a given property.

Figure 1.1 shows the process of property-based testing. To test the source code of a program, TASPEC specifications from a variety of sources are used. Program-independent specifications include system call, security, and generic flaw specifications. If necessary, program-specific specifications can also be used. The Tester's Assistant analyses and tests the code with respect to the specifications. Three results of the property-based testing process are: the test suite, the coverage results, and/or flaws discovered during the test.

This methodology for property-based testing does not include all aspects of the testing process. Other aspects, such as test case management, are left to other work. Another specific and important area that property-based testing does not address is test data generation. Some initial data is assumed to exist, the program is executed on this data, and the results are analyzed. This analysis shows where new test data is needed to fill coverage gaps; each coverage gap corresponds to a path through the program that needs to be tested. To go from the analysis to actual test data requires test data generation tools. This dissertation assumes that some method exits to derive test data based upon the analysis. In the worst case, this role can be taken by the human tester. The technique proposed as an extension to the property-based testing to address this need is symbolic execution [5] as by Demillo [11] and Korel [35].

1.3 Main ideas

The main idea and result of this dissertation is property-based testing. Specific ideas and results are:

1. Property-based testing uses the specification of properties to define validation goals, and structural analysis and testing of programs to assess the fulfillment of those goals. Structural testing is used to assure that the actual source code of the program adheres to the property specification. Validation of a property for a given program attaches semantic meaning to the program. Property-based testing is a significant addition to any testing methodology requiring formal rigor in test results.
2. Many properties are defined independently of a specific program, and so can be grouped together in libraries of properties. These libraries form models of system behavior, which are significant analytical objects in their own right. They can be reused and also analyzed by independent means to assess their completeness[1].
3. Program slicing, the extraction of all code affecting conformance to a property, reduces the amount of code that a human tester must inspect manually. Applying automatic analysis tools to the slice rather than to the whole program also aids the analyst. To calculate a slice, detailed global dependencies need to be derived; this information is used in generation of the coverage metric as well.

[1]Through a previous iteration of property-based testing, perhaps.

4. Iterative contexts, the new coverage metric developed for property-based testing, is a practical and relevant metric for satisfying property validation requirements, and iterative contexts are more powerful than other data-flow metrics. Given a set of variables at a point in the program that are of interest, the optimal metric requires all possible results for that set of variables; for most sets this requires an infinite number of data values. Metrics based upon sequences of assignments within the slice approximate this optimum for given programs.

5. TASPEC, the specification language used in the Tester's Assistant, and developed specifically for property-based testing, has primitive constructs which enables it to be translated automatically into slicing criteria and test oracles. TASPEC includes basic logical and temporal operators, together with location specifiers. Location specifiers allow events to be associated with code features. This provides the primitive data for analyzing higher-level semantic features of the program. TASPEC is a flexible low-level specification language well suited for specifying a wide range of properties and deriving tests from the property specifications.

6. Automatic high-level execution monitors which serve as test oracles are derived automatically from property specifications in TASPEC. Primitive elements of the specification state are produced via location specifiers and higher-level elements are raised by the execution monitor as dictated by the property specifications. Therefore, checking the adherence of a program execution to a complex property specification is automatic.

7. The Tester's Assistant is an implementation of property-based testing. The static analyzer that produces global dependency information about programs is completed. A slicer based upon the information produced by the static analyzer is also implemented. Portions of TASPEC and iterative contexts are also implemented. Implementation of additional features and experimentation with the Tester's Assistant and property-based testing is ongoing work.

1.4 Organization

Chapter 2 contains a sample session with the Tester's Assistant. The Unix program ftpd is tested with respect to an authentication property.

Chapter 3 puts the work of this dissertation in context with other previous and ongoing work. A rationale is given for the property-based testing approach based upon this contextual information.

Chapter 4 introduces TASPEC, the specification language which is used in the Tester's Assistant. The power and flexibility of TASPEC is illustrated by expressing several generic properties.

Chapter 5 describes how static analysis is performed and used in the Tester's Assistant. Static analysis creates a data-flow graph of a program, which serves as the basis for other analyses. One such use is obtaining a slice of a program with respect to a given property.

Chapter 6 describes the degree of program complexity addressed by various coverage metrics. It compares old coverage metrics to the new proposed coverage metric, iterative contexts. The chapter also discusses implementation issues in the Tester's Assistant for the new metric.

Chapter 7 describes the basic process of monitoring program execution. One goal of monitoring is to monitor the correctness of an execution. Correctness monitoring is done with respect to TASPEC properties. A second goal is to monitor coverage metrics. Coverage monitoring produces a set of paths which have not been executed and which can be targeted for further analysis.

Chapter 8 documents the present state of the Tester's Assistant implementation. This chapter describes the Tester's Assistant's current status, how it is operated, and experiences gained during the implementation process.

Chapter 9 summarizes the results of the work described by the dissertation and its main contributions to software testing and analysis. Future work in property-based testing is presented.

Chapter 2

Property-based Testing Session

This chapter describes testing of a version the Unix ftpd (file transfer protocol daemon) program with property-based testing. Property-based testing has eight steps:

1. Selecting a property; property is specified in TASPEC (currently implemented)
2. Static analysis and slicing (currently implemented)
3. Program instrumentation (currently implemented)
4. Initial test case selection and execution
5. Coverage evaluation
6. Additional test case selection and execution
7. Correctness evaluation
8. Repeat the last three steps as necessary

Property-based testing ftpd with respect to an authentication property reveals a flaw in ftpd's authentication code. Only the initial three steps of property-based testing have been implemented.

2.1 Description of ftpd and its flaw

Ftp is a Unix program for transmitting files across a network. Ftpd, the program described here, is a server program that accepts file requests, processes authentication and other utility commands from remote client programs.

In the version of ftpd released with Sun Unix 3.2, a security flaw allows any user to gain permissions to read or write files owned by any user on the system (including

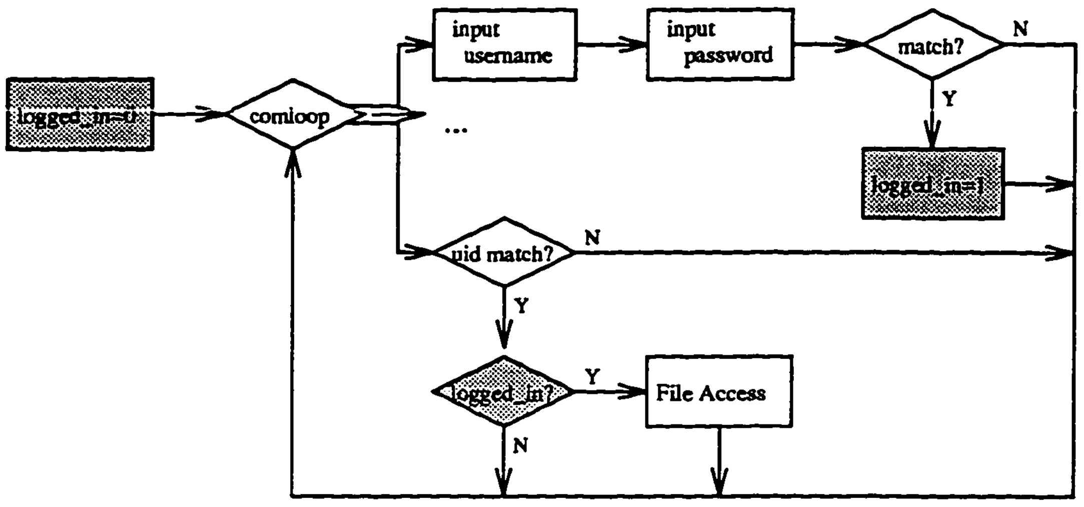


Figure 2.1: Ftpd flaw flowchart.

root) [7]. To do so, the user logs on with his or her normal user name and password. As a part of the correct authentication, a flag in the program is set. The flag records whether the user name has been authenticated. When a second user name is entered, the flag is never reset, so even if an incorrect password is entered for the second user name, the program thinks that the second user name has been authenticated. Therefore, the user has the access privilege of the second user name. Figure 2.1 is a simplified flow-chart displaying the flaw.

2.2 Selecting/identifying a property

The first step in property-based testing is choosing a property or properties from a selection of generic properties. If there is a specific program-specific property to test, then that property should also be written. Property specifications are written in TASPEC, a specification language designed for property-based testing. In the case of ftpd, a generic property is used.

A portion of the property library is a set of properties which describe a security model. These properties use some of the same low-level properties and so are related to each other. One high-level property specification requires that authentication occur before any permissions are granted:

$$authenticated(uid) \textbf{ before } permissions\_granted(uid).$$

The library also contains low-level definitions of the predicates *authenticated* and *permissions_granted*. The TASPEC language and elements of the property library have been designed and written. Section 4.6.1 presents the complete specification.

The authentication property can be selected by hand. Optionally, an automatic tool could compare location specifiers (code templates) in the property specifications with the source code of ftpd to evaluate the relevance of properties in the library. The definition of *permissions_granted* involves the `setuid` system call[1]. The property, then, forms a pre-condition for the `setuid` system call. Since ftpd contains `setuid`, then the authentication property can be automatically chosen as an important property for which to test.

2.3 Static analysis and slicing

The Tester's Assistant statically analyses the source code for ftpd. Ftpd contains about 3000 lines of C code, 1700 lines of which are machine-generated. The static analysis produces a data-flow graph for ftpd. The ftpd data-flow graph has 6148 nodes and 31912 edges. The data-flow graph is used in other steps of the process: program instrumentation, coverage evaluation, additional test case generation, and correctness evaluation.

Next, slices of ftpd are derived from the data-flow graph. First the slicer generates a slice of ftpd with respect to the selected authentication property. The human tester inspects the slice manually, but even in the sliced code (represented in Figure 2.1) the flaw is subtle enough that it goes unnoticed. At this point the human tester can request additional slices based upon any other criteria that can aid in the tester's understanding of ftpd.

[1]`Setuid` is used here as an amalgam of the many different permissions-setting system calls (seteuid is used by ftpd).

2.4 Program instrumentation

The Tester's Assistant produces an alternate version of ftpd, that will be the actual version executed during testing. The alternate version has the same functionality as ftpd, but additionally has data-gathering modules, so that coverage and correctness can be evaluated from test results. Every section of source code corresponding to a location specifier in the property has code added to record if and when the section of code is executed. The added code is used later in correctness evaluation. The assignments in the source code which are significant for coverage evaluation are also tagged to record when the assignments are executed. The Tester's Assistant instruments only the slice relative to the selected authentication property. The instrumented source is then compiled, at which point ftpd is ready to be executed.

2.5 Initial test executions

The instrumented ftpd is executed several times with various test data. There are three ways to generate test data for ftpd: First, any available test data that was used in initial testing and debugging by the developer of ftpd can be used at this step. Second, simple test data can be generated by the tester from a description of ftpd's functionality. Finally, if there are any specifications of ftpd, the specifications can be used to generate test data. Generating test data from specifications is not specifically part of property-based testing, but other testing methodologies contain the necessary algorithms [8, 12, 55].

The first method is simplest, because no extra work is required and the test suite is likely to be fairly complete. However in this case, the test cases aren't available, so the human tester creates some test cases by reading the ftpd manual page. Figure 2.2 shows some sample test cases.

The test executions are then evaluated for coverage and correctness. None of the four executions result in a violation of the authentication property. However, coverage evaluation reveals that ftpd has not been completely tested, so more test cases must be found and executed.

```
user <user name>
pass <incorrect password>
retr filename

user <user name>
pass <correct password>
cwd  directory
retr filename1 filename2
```

```
user <user name>
pass <correct password>
retr filename (no access permissions)

user <user name>
pass <correct password>
list
```

"User" enters a user name, "pass" enters a password, "retr" retrieves a file, "cwd" changes directory, and "list" lists a directory.

Figure 2.2: Four initial test cases for ftpd.

2.6 Coverage evaluation

While ftpd executes with each given test data, the coverage instrumentation writes a file recording the execution history of the slice. The execution history indicates which path in ftpd was executed. The initial test executions yield several execution histories. The execution histories are compared with the coverage metric. Property-based testing uses a new metric, iterative contexts. Each context is an ordered sequence of assignments, which defines a sub-path of the program. For a history to match a context, the assignments must be executed in order with no intervening and interfering assignments. The contexts are generated using static analysis and the data-flow graph,

For the (abstracted) fragment of ftpd source

```
(1) logged_in = 0;
(2) while(1)
(3)   switch(cmd) {
(4)     user:  name = read();
(5)            pass = read();
(6)            if(match(name,pass))
(7)              logged_in = 1;
(8)            break;
(9)     get:   if (logged_in)
(10)              setuid(name);
(11)    }
```

the contexts required include $\{\{4, 5, 6, 10\}, \{4, 5, 6, 4, 10\}, \{4, 10, 4, 5, 6\}\}$.

The execution histories are compared with the set of contexts to see which histories match which contexts. The contexts which are not matched are coverage gaps.

The execution histories from the four initial test cases are

$$\{\{4, 10, 4, 5, 6\}, \{4, 5, 6, 10\}, \{4, 5, 6, 10\}, \{4, 5, 6\}.$$

The second and third execution histories are identical because their behavior relative to the property specification is identical. The context $\{4, 5, 6, 4, 10\}$ is a coverage gap in the initial test data, and corresponds with the flaw in ftpd.

2.7 Additional test cases

In order to complete the coverage metric, additional executions of ftpd are necessary, with different test data that addresses the coverage gaps. This dissertation does not present a method to produce this additional test data automatically, and the problem is not trivial.

A human tester produces additional test data by examining the contexts not covered and the code corresponding with the contexts. For the contexts and code in ftpd, there is a close correspondence between input statements and statement numbers in the uncovered context (Statements 4 and 5). The uncovered context $\{4, 5, 6, 4, 10\}$ is executed by the the following test script:

```
user <user 1's name>
pass <user 1's password>
user <user 2>
pass <random string>
retr  filename
```

Correctness evaluation of this execution detects that the flaw exists in ftpd.

Future versions of the Tester's Assistant may be able to automate some of the steps in generating test data for gaps in coverage using techniques based upon symbolic execution [11].

2.8 Correctness evaluation

During each test execution, a file is written recording the TASPEC primitives activated during the test. The TASPEC evaluation engine processes this data and compares it with the property specification. If the data violates the property specification, then the human tester is informed that the test caused an error condition.

During processing of the correctness records for the additional test case given in Section 2.7, the correctness monitor registers that there is a correct authentication of user 1. No authentication of user 2 is registered, because the password match fails. When the file retrieve action occurs, the *permissions_granted* property is registered. However, the retrieve occurs with the permissions of user 2, for whom there is no authentication. Therefore, the additional test case causes an error condition, so ftpd fails the property-based testing with respect to the authentication property.

Chapter 3

Background, Methods, and Examples

This chapter describes the background and rationale for property-based testing. Section 3.1 introduces the computer security issues that motivate property-based testing, and also introduces the three programs, `ftpd`, `rdist`, and `fingerd`, that are used as examples in this dissertation. The remainder of the chapter covers the background for testing methods used in property-based testing. Section 3.2 discusses specification. Section 3.3 reviews static analysis and the issues involved in static analysis of C. Section 3.4 defines program slicing, briefly describes other work in the area, and indicates how slicing is applied to property-based testing. Section 3.5 defines the terms *coverage* and *metric*, and introduces issues with coverage metrics addressed by property-based testing. Section 3.6 relates debugging tools to correctness monitoring in property-based testing.

3.1 Computer security and property-based testing

Assuring that computer programs and systems are secure is an important and difficult problem. Security flaws are still being discovered in computer programs that have been in use for many years. Many of the flaws are caused by the same basic recurring faults [60]. For example, the Internet worm [59] exploited errors in Unix network programs. Examination of the flaws which caused the errors revealed them to be of an elementary nature.

It is time for a concerted effort to try to prevent such flaws from occurring, i.e., with property-based testing. Therefore, an appropriate initial application of property-based testing and the Tester's Assistant is Unix security, specifically for network programs. Security is a good application of property-based testing because the parts of programs that relate to security are small, and generic security properties can be precisely expressed program-independentally with TASPEC.

3.1.1 Issues in computer security

Networked systems cause special security problems because any communication or authentication between networked systems must be performed entirely through an exchange of information. The exchange of information is limited by the network structure as well; many networks in use today are asynchronous, and make no strict delivery guarantees for information packets. Problems with asynchrony are complicated by different implementations for the same service protocol, which may have different performance. Therefore, network services must be flexible in their implementation of communication and authentication services. This flexibility can sometimes be exploited and become a source of security problems, adding to security problems arising from bad design or implementation.

Network services with Unix involve the client/server architecture. The server runs on a host machine, and regulates access to information on the host by communication with server processes on other machines on the network. The server can do its task in one of two ways: by forking off a server-end client process to handle commands, or by doing all the work internally. In either case, the server will be interacting with the host system in a number of ways – reading/writing files, etc.

Most network servers are privileged programs; they are run with special root privileges on the host machine. Unix has a coarse-grained tri-level file protection scheme. If the access level for a process cannot fit into this scheme, the process must be given root level permissions, which override the scheme. Network services typically do not fit into the tri-level scheme and are given root permissions, even though root permissions are used in only one particular function of the program. Therefore, the server is given excess privileges, which become fertile ground for exploitable vulnerabilities.

3.1.2 Example programs with security flaws

Three network programs for the Unix operating system are used as examples here. **Ftp** is a program for doing file transfers between the local machine and a remote machine on the network. It is described in detail in Chapter 2.

Rdist is a Unix program designed to update file systems on distributed hosts. A master file structure is set up on the server, and when rdist is run, the corresponding remote file structures are checked to see if any of them are out of date. If so, the file is copied from the master[1], and rdist checks other machines for outdated copies, replacing them as necessary. Rdist invokes a copy of itself on the remote machine and establishes communications between the master (local) rdist and the slave (remote) rdist. The master then transmits commands to check and receive files to the slave rdist. In order for rdist to maintain files owned by many users, rdist is run with root permissions, so it can perform any file operation (such as changing ownership and protection attributes).

The slave portion of some versions of rdist has a race condition flaw [6]. The target file is copied into a temporary file and only when the file is completely written is it changed to its correct location and correct ownership. A user can replace the temporary file while it is being written and thus cause the wrong file to be moved and have its ownership changed. Through this method, a user could claim ownership of the password file, for example, and change passwords so that super-user access is possible. The flaw is called a *race condition* because rdist and the user "race" to access the temporary file. Testing for exploitable race conditions catches this flaw.

Finger is a program for getting information about users on a machine across a network. Finger takes two arguments: a machine to check, and a user to check for.

[1]It is also possible to use rdist to update separate copies of the same file on the same machine. However, using rdist on distributed hosts is more typical.

A connection is made with the remote machine and the user name is sent across the connection. The remote machine reads the user name and sends back the user information. In some versions, the remote machine did not check the size of the user name, but read the user name into a fixed size data space on the stack. By supplying a very long user name, the user could cause an array overflow on the remote machine. The portion of the name that overflows onto the stack could contain code that is executed by the fingerd process. Therefore, using finger, a user could execute any arbitrary code on the remote machine. This flaw was exploited by the Internet worm [59]. A test which addresses array boundary conditions catches this flaw.

3.1.3 Approaches for establishing security

There are two basic ways to provide security for a computer system hooked up to a network:

1. Actively try to detect intrusions and other insecure behavior and take corrective action.

2. Prevent intrusions by using secure software and secure configurations.

Preventing intrusions is more desirable than detecting them; when insecure behavior has been detected, damage may have already been done.

Either detection or prevention uses a model (either explicitly or implicitly) of secure and insecure behavior. This model describes a "security policy" to which all programs in the system must adhere. The security policy is the definition of what is and is not secure. A typical element of a security policy is a description of file ownership by different users and how access is regulated, including some notion of how different users are identified and authenticated.

The Unix machine consists of relatively few object types (kernel, user, process, file), which simplifies system modeling, an important issue for property-based testing (and for any other validation methodology). Without some concept, embodied in a model, of the types of security issues present in a system, there is no hope of detecting or preventing security problems.

Consider the following hypothetical flaw. Suppose the memory image of a program is not zeroed out when it terminates, so that its memory can be reallocated to another process. A (malicious) process could request large amounts of memory, and scan the memory it receives for pre-existing data. If a previous (privileged) process left clear-text login names and passwords in its memory image the malicious process could use that information to obtain unauthorized access. If this flaw can be represented in the system model, then it can be detected and fixed by requiring memory to be cleared before a process exits.

A common prevention method is informal, or "ad hoc" testing. Ad hoc testing programs for security flaws includes a variety of techniques, but usually one or more "tiger team" members examine the code by hand or using simple tools. They look for known patterns of unsafe code; should any be found they attempt to build attack scripts which cause security violations when the code is executed. Both this model of unsafe code and the skill and knowledge of the individual analyst limit the effectiveness and completeness of this technique. A computer program can be more thorough, systematic, and complete; such is the function of the Tester's Assistant. In addition, the need to formalize a security model means it too can be examined for consistency and completeness.

Verification (formal proof) of security properties has the advantage of a formalized model, but case studies of verification in have shown that programs which are not written with verification in mind are difficult if not impossible to verify [23]. Property-based testing, in addition to being a technique which can be utilized by relatively untrained users, has been designed with the understanding that third-party code would regularly be its subject. Third-party testing is possible because testing is based upon generic program-independent specifications.

The problem of overly privileged network servers could be addressed by intrusion detection [47, 58] or other run-time methods such as audit trail analysis [46]. For this defense to be effective, wrappers and intrusion detection must always be invoked; depending on the circumstances if an error is detected the damage may already have been done. A preventive measure such as property-based testing prevents even this first security violation.

3.1.4 Using property-based testing for security

The approach described in this dissertation is to use formal testing to validate security properties of software and thus produce secure systems. Security-related code is often only a small part of a program's functionality. Property-based testing focuses on code relevant to security functionality in great detail, and so efficiently validates the security-related part of the program without testing the whole program.

Property-based testing provides a system and methodology for testing narrow properties of source code. It produces a specific and absolute metric for successful testing with respect to those properties. A successful test validates that properties are not violated; if these properties form the security policy for the system, then the system is secure.

Property-based testing uses a security model of the system, as well as a library of generic flaws (such as [38, 60]) specified in TASPEC, to produce a test process, whereby the target program can be certified to be free of certain types of flaws.

3.2 Specifications

Specifications are descriptions of software (or hardware) systems. Specifications describe what a system should or should not do; they can be written in whatever language is most appropriate for the task. For example, specifications written in English can be easy to read and understand, but ambiguity and lack of formality make it difficult to systematically map English specifications to software systems. Diagrams such as flow-charts are graphical ways to express specifications. Specifications in formal languages such as Z [13, 61, 67] are easier to systematically (but not automatically) relate to computer programs, and can be used with formal reasoning tools, but can be harder to produce and understand.

The basis of property-based testing is specification of properties. The property specification drives testing in an automated fashion. Therefore, the specification of the property must be concrete in that elements in the specification language must be explicitly related to source code structures. However, the specification of the property must also be generic and not program-specific, so that a single specification can be used in tests of many different software systems.

The specification of a property is not meaningful in isolation. For most properties it is necessary to place the specification as a part of a larger model (or security policy). One of the examples used throughout this dissertation is authentication. Authentication is one aspect of a larger security policy. Property specifications of authentication are shared with other specifications of that policy; e.g., that correct authentication must precede the granting of access to a system.

The security policy forms a higher-level abstraction model which can be examined for internal flaws and defects independently from the tests of specific programs. Abstraction is one of the major powers of specification. Lower-level details are abstracted from a model so that only the relevant high-level abstract behavior, represented in an appropriate abstract form, can be examined. Property-based testing validates the individual property specifications; other analysis can use these validations to make assertions about the overall policy implemented by the system.

3.2.1 Specifications and testing

Specifications describe models that programs are supposed to implement. Testing is used to verify if a program's implementation is correct. The goals behind using specifications in testing are establishing greater formalism for test results, and increasing the automatability and re-usability of test objects. Earlier methods for utilizing specifications in testing fall into three categories: Specifications to generate test data [57, 8], specifications to create test oracles [18, 17, 53] (verifying the correctness of an execution), and specifications refined into code [26, 16, 33, 14] (and therefore having direct measurable specification-code relationships).

Specification languages such as Z [13] and VDM [2] can be used to fully specify a system at a level more abstract than source code. Data structures and operations can be gradually made more concrete through refinements to the specification, until at some point the boundary between specification and source code is crossed. Presumably, the more abstract specifications better reflect the desired abstract functionality (though they are less specific), so concrete execution states can be compared to abstract states via the correspondences extracted from the refinement process. This comparison can serve as a test oracle. Such a specification could also be used to generate tests. However, generating test cases from such a specification is not very different from generating tests from the source code due to the shared derivation of code and specification.

Test oracles can also be automatically generated from other specification languages such as Larch [24] and TAOS [54, 53]. Function and procedure behavior is specified as in the refinement methods, but in a separate process from the actual coding. The specifications can then serve as independent test oracles without being influenced by implementation bias. Links between specification objects and implementation objects need to be provided so that the respective states can be compared. This linkage can be made easily if the unit of specification is the behavior of individual functions (or modules) in the implementation. Formal parameters can be linked with actual parameters, and so on.

Test data can also be generated from specifications. ADL [56, 57] and TAOS have test description languages by which test data can be partitioned. Once the input domain is partitioned, generating exhaustive test data with respect to the structure of the specification is possible [55, 8]. Prior to this work, similar techniques had been used with VDM [12]. Related work generates test data from the structure of code such as in [21] and [29].

Using location specifiers, generic program-independent properties in TASPEC map automatically to source code. Therefore, test oracles can be generated independently of descriptions of specific modules or functions. With the emphasis on *properties* and not on full specification, test oracles can be made to handle a wider class of behavior than that rigidly defined by functional specifications. Translations between other specification languages and TASPEC can provide additional flexibility to the specification and testing phases of development. Helmke shows how translations from Z to TASPEC can assist in requirements traceability [26].

3.3 Static analysis

Static analysis of source code is the basis of most formal software testing [25, 36, 39, 9, 49, 15, 27, 45, 51, 65]. Many static analysis techniques use the relationship between an assignment to a variable and the usage of that variable later on [25, 39, 9, 49, 27, 65]. This dependence, called a data-flow dependence, captures the flow of information in a program. Static analysis typically requires the calculation of all possible data-flow dependences. During any single execution of the program, not all of the program code is executed, so many of these potential dependences are not actual dependences for that execution. Determining which are the actual dependences for an execution is not possible statically; however, static analysis can find all alternatives. The extent to which static analysis can reduce alternatives statically reduces the number of test executions required to examine the remainder.

Static analysis is also used in other programming language analysis contexts, such as compilers [19, 1]. In general, compilers have weaker requirements than formal software testing tools. Compilers can, in most cases, function with only local data-flow analysis. The information needed for code scheduling and register allocation can be calculated locally. For precise testing results, it is necessary to thoroughly analyze global data-flow relationships. A good example of the difference between compiler analysis and testing analysis is function side effects. A compiler can simply refuse to schedule code across function calls, so that any side effects from the function do not affect the code around the call site. But a testing environment needs to know about those side effects and their effects on the current state. Therefore, static analysis for a testing system must do full global data-flow analysis[2]. The remainder of this section deals with global data-flow-based static analysis techniques used in software testing.

The basic data structure is the control flow graph. Each block of statements forms a node in the control flow graph. Edges represent direct flow of control: if $\mathbf{block_1}$ could execute immediately after $\mathbf{block_2}$, then there is a corresponding edge in the control flow graph. The control flow graph is intra-procedural only; an inter-

[2]More recent optimizing compilers also use global data-flow analysis, e.g., for inter-procedural analysis.

procedural control graph of an entire program is too large and unwieldy for effective analysis. Inter-procedural analysis uses summary information generated from the data-flow graphs of individual functions.

The control flow graph is augmented with dependence edges to form the program (or procedure) dependence graph (PDG [51]). If $\mathbf{node_1}$ uses a value which is generated at $\mathbf{node_2}$, then there is a data dependence link between the two nodes. If the execution of $\mathbf{node_1}$ is dependent upon a predicate at **node2**, then there is a control dependence link between the two nodes.

The presence of functions complicates calculation of the PDG. The data-flow behavior of a function depends partially upon the data-flow behavior of functions which it calls. Additionally, a function can be invoked from many different locations in the program, with different types of input parameters. The body of the function needs to be processed and then summarized so that the result of the function's data-flow analysis can be applied to the different calling locations.

Additionally, the typical language processor can process one input file at a time. For testing, global data-flow analysis is necessary, so at some level all files need to be processed simultaneously, in case there are recursive dependences between files. In practice, because a parser can only deal with one input file, a static analyzer needs to decide which file to process first and to completely summarize the data-flow properties of each file in turn.

Data-flow graphs are used to produce program slices and coverage metrics. Chapter 5 has details about static analysis and production of data-flow graphs in the Tester's Assistant.

3.4 Slicing

Definition of slicing

Let *P* be a program, let *p* be a point in *P*, and let *V* be a subset of the variables of *P*. A static slice or simply a slice of *P* relative to the *slicing criterion* $<p, V>$ is defined as the set of all statements and predicates of *P* that *might* affect the values of variable *V* at the point *p*. [64]

```
int example (int x, int y)          int example (int x, int y)
{                                   {
 int z;
 z = 0;
 if (x == 0)                         if (x == 0)
  z = x + y;
 else                                else
  x = y;                              x = y;
 if (y)                              if (y)
  setuid(x);                          setuid(x);
 else
  return z;
 return -1;
}                                   }
```

Figure 3.1: Simple example function before and after slicing.

Slicing is the most important technique used in property-based testing. Program analysis algorithms such as data-flow coverage tend to be at least exponential in running time [9] (if not uncomputable in the general case). No testing methodology can hope to scale to large software systems (or even to medium-sized programs such as Unix system code) using such algorithms, unless the scope is sharply limited. Slicing eliminates source code which is irrelevant to the property, thereby making feasible the execution of generalized algorithms over the remaining code.

An ideal static slice is a minimal representation of the computations which might be involved in the evaluation of a certain variable at a certain point in the program. To calculate slices, complete control- and data-flow information about the program is needed. Once this information is known, slicing can be done in $O(n)$, where n is the number of nodes in the data-flow graph (which is directly proportional to the size of the program). Specifically, once a correct and complete data-flow graph is constructed, a single graph traversal calculates the transitive closure of a set of nodes with respect to certain types (and directions) of edges. Section 5.4 gives more information about slice construction. An example of slicing is in Figure 3.1. A static slice with respect to the variable `x` parameter of the `setuid` function call is shown alongside full example function.

3.4.1 Uses of slicing

Slicing is used in software maintenance [20, 41, 63], automatic parallelization [64], and testing and debugging [64, 63, 37]. In software maintenance, it is useful to examine the side effects of a single change to the source code. Slicing on this point in the code, then, produces a map of exactly where else in the code more alterations may be needed. Slicing has additional applications to maintenance [20]. Other slicing work is described in [42, 43, 3, 15, 63]. Slicing is an active area of research and development.

For automatic parallelization, output variables in a program are identified, and slices are computed for each output variable. Each slice can be executed separately. Thus, if the slices are smaller than the original, and every slice can be executed concurrently, the running time of the whole program can be reduced, assuming the overhead for calculating slices and starting multiple processes to execute the slices is not high. Weiser tested these ideas with a slicer for a version of Fortran [64]. He also proposed that a slicer would be useful in debugging, by slicing when an error occurs to find possible faults in the program.

STAD [36, 37] uses a related concept called a dynamic slice to debug programs. Each dynamic slice is calculated with reference to a specific execution of the program, which means that the slice consists of all statements of the program that affect the values of the target variable (the "might" in the definition of slicing is dropped). This distinction allows the slices to be much smaller than the classical static slice. With this smaller slice, it is much easier to identify the exact cause of the program flaw being examined.

The Tester's Assistant uses slicing in ways similar to these. First, the simple inspection of a slice can be instructive to a tester. The slice can reveal an unanticipated dependency, which is sometimes indicative of a flaw that could be found even before testing commences.

Second, a slice can be tested separately from the rest of the system. If by slicing with respect to security properties the security-relevant portion of the program has been isolated, then only those inputs which have bearing on these portions need be tested. Test execution will be faster because the extraneous computations have been removed. The observed behavior of the program will also be limited to that which is of interest. (See Section 5.4.)

Third, a whole program could be executed, with only those portions which are in the slice monitored. Since monitoring is an expensive operation, computational efficiency is again improved by limiting monitoring to just the code pertaining to the property of interest. (See Chapter 7.)

Fourth, a slice can be used in the computation of the completeness of a test with respect to the slice's property. Rather than using some global path- or statement-based test coverage criterion, the criterion can be based upon the code within the slice. The number of paths in a program is exponential in the number of branches of the program (assuming some fixed way of counting loop-induced paths). If a slice is an order of magnitude smaller than the original program, the corresponding reduction in the number of paths which need to be tested is dramatic. (See Chapter 6.)

3.5 Coverage analysis

A fundamental issue in testing is determining when testing is complete. Typical specification-based testing schemes base this determination upon how thoroughly the test data reflects the intricacies of the specification. Typical code-based testing schemes base this instead on the intricacies of the code. We concentrate on code-based schemes in property-based testing, with some modifications.

The basis of code-based test completeness is coverage. A particular segment of code is covered if it is fully executed in a test case. Coverage is calculated with respect to some model of what constitutes sufficient testing; differing models have different coverage criteria. For example, in statement coverage, $a = b/c$ is "covered" if it is executed once. Operator coverage is similar to statement coverage but includes special cases for arithmetic operators, so test data of $c = 0$ (a special value for the second operand of the division operator) would be a second, required, case. The public domain coverage analyzer GCT [48] implements statement and operator coverage for C.

A common model for coverage is that of definition-usage (def-use) pairs. $< d, u >$

is a def-use pair if d is a definition (assignment) of some variable, and u is a usage of that variable. One version of def-use coverage might require d and u to be executed in the same run of the program. Def-use pairs are used in many metrics [65, 9, 39], and in one public domain coverage analyzer [27].

An alternate model for coverage, not utilized in the Tester's Assistant, is mutation testing [11]. Mutation testing distinguishes between the the program and closely related programs, each related program differing in only one small way from the original. The measure of completeness is the number of distinct, but related programs covered by the test suite. Consider the example used above, $a = b/c$, and specifically the operator /. Assume that the code fragment is actually in the program to be tested, but not necessarily in the correct version of the program. If this operator is in error, then in the correct version of the program, / could be any of the related operators $+$, $-$, etc. So a test suite would need to supply values for b and c such that the value of a after this assignment would be different depending upon the operator used.

None of these metrics are strong enough to catch the ftpd flaw in Chapter 2. This dissertation introduces *iterative contexts*, an extension of def-use schemes that is strong enough to catch the flaw. Chapter 6 presents def-use and iterative contexts in more detail.

Coverage models and metrics can be compared in terms of relative strength. A metric M supersedes (i.e., is stronger than) a metric M' if any test set that satisfies M also satisfies M', but not vice versa. Chapter 6 describes a series of coverage metrics, and relates them to each other in terms of strength, building on work by Hamlet [25]; this description motivates iterative contexts. As a result of Hamlet's analysis, he identifies what he labels the "locksmith" problem.

The locksmith problem involves two adversaries, a thief and a locksmith. The locksmith tries to design a lock which the thief cannot open. There is no foolproof lock, just locks with better and more complex schemes to foil lockpicking. The thief can keep coming up with more sophisticated ways to break the lock, and there is a never-ending procession of one-upsmanship.

Computer security has a similar problem, and to some extent, program analysis and formal testing do as well. Tests can be formulated which can catch a certain kind of flaw, but there are likely to be more complex flaws which current tests do not catch, necessitating development of improved tests, and so on, ad infinitum.

3.6 Debugging

As errors are found through testing, debugging tools are used to trace the source of the fault. The Dalek debugger [50] uses a powerful language to monitor and control the behavior of the program being debugged. The concept of event handling as a monitoring and debugging tool is a feature of the Dalek debugging language which inspired the use of event handling in the Tester's Assistant.

In Dalek, a primitive event is tied to a breakpoint in the code. A breakpoint can be broadcasted so that it is attached to all code of a specific type, for example, all invocations of a specific function. Dalek encompasses an event engine that allows semantic processing of primitive events. Through the event engine higher-level events are "raised", and are fed back into the event engine. By using this functionality, an individual debugging a program can track higher-level behavior and more readily find the source of a particular error.

The Tester's Assistant's run time correctness monitor uses TASPEC to tie events to code, and complex events to primitive events. The event stream is processed by a Prolog-like engine which can make inferences and detect flawed execution. See Chapter 7 for more details.

Chapter 4

TASPEC

TASPEC is a new specification language that serves as the specification interface for the Tester's Assistant. TASPEC can represent general properties of programs and system environments, and is simple enough so that test oracles and other testing mechanisms can be derived automatically from TASPEC specifications. TASPEC has an event-driven framework; events in the execution of a program drive instantiation of TASPEC objects. TASPEC constraints are compared with the TASPEC run-time state with Prolog-like inference [10].

TASPEC can express close correspondences between code and abstract semantics, which enables their translation into execution monitors and slicing criteria. TASPEC includes basic logical and temporal operators, as well as location specifiers that associate events with code features. Location specifiers provide the primitive data needed to analyze the abstract semantic features of programs.

This chapter defines the TASPEC language and gives several examples of its use to describe generic flaws. Section 4.1 shows the relationship between property specifications in TASPEC and a program being tested. Section 4.2 describes the basic concepts in TASPEC. Section 4.3 defines the TASPEC language. Section 4.4 outlines implementation issues with TASPEC. Section 4.5 gives a completeness result for TASPEC. Section 4.6 gives three examples applying TASPEC to generic flaws. Section 4.7 summarizes some positive and negative aspects of using TASPEC as a specification language.

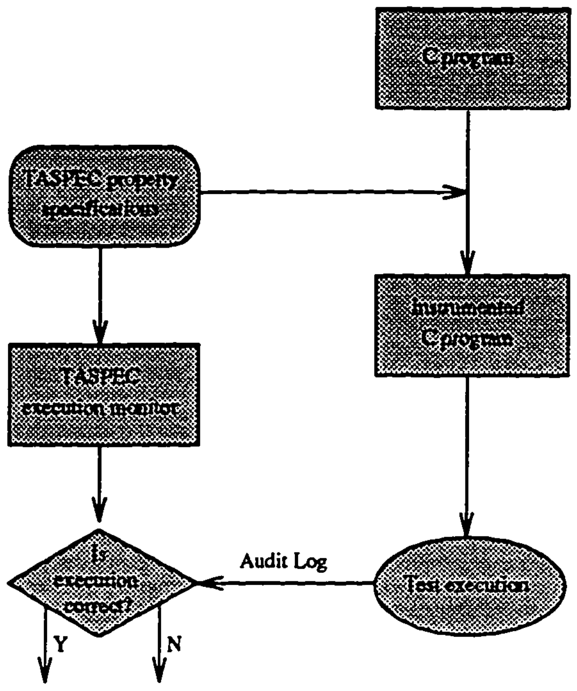


Figure 4.1: TASPEC properties used in the test of a C program.

4.1 Overview

The primary application of TASPEC property specifications is evaluation of the correctness of an executing C program (Figure 4.1). First, the specifications determine the instrumentation of the target C program. The instrumented program contains additional code that emits an audit log of locations specific to the TASPEC properties. Second, an execution monitor is derived from the property specifications. The execution monitor evaluates TASPEC events and constraints based upon the audit log. If no constraints are violated by the audit log, then the execution is correct.

During analysis of the audit log, the execution monitor uses a database of predicates to record the abstract state of the program execution. This state changes as each new event from the audit log is processed. Every time the state changes, the state is checked for consistency with the constraints from the original specifications. The creation and evaluation of the abstract state can occur either after the execution terminates, or in real time as the program is executing; choice of the time of evaluation does not affect how the evaluation is performed.

In addition to correctness evaluation, TASPEC properties drive slices (Chapter 5) and calculate coverage metrics (Chapter 6).

4.2 Basic TASPEC concepts

TASPEC specifications define an abstract state model and how the model is instantiated into an abstract state by the execution of a program. The abstract state is defined by two types of specifications: predicates that form the state, and events that operate on the state. The three event types are assertions that change the state, invariants that form constraints on the state, and pre-conditions, which are invariants that if true activate additional events. Special pre-conditions called location specifiers are templates which match source code; when the source code is executed, the pre-condition is activated.

Location specifiers are used to instrument a program. Then, during the execution of the program, the abstract state specification is transformed into a concrete stream of predicates. The state stream is checked against the invariants in the abstract state in order to detect errors.

TASPEC's semantic model is loosely based upon that of Prolog. The abstract state consists of a collection of predicates tied together by various invariants. The language for expressing invariants is closely related to Prolog, as the system was designed with Prolog as the engine for monitoring execution. More details about the execution model are in Section 4.4.

The basic state descriptor is called a predicate. For example,

$$permissions\_granted(287).$$

Predicates are entirely defined by the specifications; *permissions_granted* is not a part of the syntax of the language. If this predicate is put into the state stream, it signifies that at that point in the execution of the program, permissions were granted for user 287. Any string can be defined by a specification as the label for a predicate, and the specification, not TASPEC, controls the string's semantics.

TASPEC specifications also describe invariants which an executing program must satisfy. For example,

$$\mathit{authenticated}(U)\ \textbf{before}\ \mathit{permissions\_granted}(U)$$

specifies that an *authenticated* predicate must be present in the abstract state at the time a *permissions_granted* is asserted, and the parameters of the two predicates must match.

Invariants are expressed by linking predicates together with standard logical connectives. Invariants form constraints which define legal abstract states. To detect errors, the invariant constraints are compared with concrete states generated through program execution.

TASPEC sub-expressions allow unbound variables. When the program is executed, the unbound variables are matched with values either directly from the executing program or from the values in the pre-condition which drive the processing of the sub-expression. For example,

$$\text{assert}\ \mathit{permissions\_granted}(U);$$

is the expression that generates the *permissions_granted* concrete state predicate.

The value U is matched with the value 287 from the context of the assert, that is from the argument to the `setuid` library call:

$$\textbf{location func}\ \mathit{setuid}(U)\{\text{assert}\ \mathit{permissions\_granted}(U);\}$$

This is an example of a location specifier: **func** *setuid*(U) matches every function call `setuid` as it is executed.

4.2.1 Simple TASPEC example

To illustrate the basic concepts of TASPEC, consider bounds checking of statically declared arrays. The array indices are declared to be a consecutive subset of the integers, i.e.,

```
declaration:  a is array[l..u]
```

When the array is referenced, a given index is used, e.g.,

```
a[i]
```

For the above declaration and reference, `i` must be between `l` and `u`, i.e.,

$$l \leq i \wedge i \leq u$$

The declaration and reference generate TASPEC state elements:

code construct	generated TASPEC state element
declaration	$array(a, l, u)$
reference	$arrayref(a, i)$

The array bounds check forms an invariant on correct TASPEC state elements. Any time an *array* or *arrayref* state element is generated, the following invariant must hold:

$$arrayref(a, i) \wedge array(a, l, u) \Rightarrow l \leq i \wedge i \leq u.$$

This specification says that in a correct state, if $arrayref(a, i)$ and $array(a, l, u)$ are in the state (the values of a must match), then the formula $l \leq i \wedge i \leq u$ must hold.

Section 4.6.3 describes in more detail the TASPEC specification for static array bounds checking.

4.3 TASPEC language description

TASPEC contains four major constructs: state elements, compounding operators, location specifiers, and event processing. This section explains each of these components in detail. A complete grammar for TASPEC is given in Appendix A. Symbols in TASPEC are pretty printed with mathematical and logic symbols; the underlying representation, however, is represented by the grammar in the appendix. TASPEC key words are printed in bold face. Other identifiers represent either names of predicates or variables, distinguishable by context.

TASPEC has limited support for C data-types. Array, structure, and pointer data-types used in location specifiers have the expected semantic meaning because they are used as text in the instrumentation that produces the values used in the evaluation monitor. Section 4.4 describes some limitations on expressing complex structures in TASPEC.

The explanation of TASPEC in this section is illustrated by the array bounds example introduced in Section 4.2.1. A more complete explanation of all the components of this specification and how they fit together is given at the end of the chapter in Section 4.6.3.

4.3.1 State elements

State elements represent the basic facts about the state of the system. Facts are expressed solely by using simple predicates, such as $array(a, 0, 9)$ or $arrayref(a, 3)$ For parameters of predicates, only the basic types int, float, and string (char *) are allowed. This design decision was made to simplify the implementation.

Parameter values of types other than those supported are converted to the fundamental integer type. For pointer objects, this representation is sufficient to distinguish between distinct objects. With this representation, if the memory address of *a* was 0x123, then the actual state elements would be *array*(0x123,0,9) and *arrayref*(0x123,3).

Multiple state elements with a common identifier can appear in the state at the same time. Most often, for each instance there are different argument values. In the array bounds example, during a test execution there will be multiple instances of the predicate *arrayref* in the state. An execution of the loop

```
for(i = 0; i < 5; i++) {
   printf(a[i]);
   }
```

would result in the five state elements

$$arrayref(0x123, 0), arrayref(0x123, 1), arrayref(0x123, 2),$$
$$arrayref(0x123, 3), arrayref(0x123, 4)$$

being added to the state[1].

4.3.2 Compounding operators

- Arithmetic operations $(+, -, *, /)$

[1]For the sake of readability, identifier *a* is substituted for its value 0x123.

- Relational operators (=,<,>,<>)
- Logic operations ($\vee, \wedge, \neg, \Rightarrow$)
- Temporal operations (**before, until, eventually**)

Arithmetic and relational operators are used to place constraints on data values. For example, consider this fragment from the specification of array bounds

$$arrayref(a,i) \wedge array(a,l,u) \Rightarrow l \leq i \wedge i \leq u.$$

The variables i, l, and u are bound by their presence in the state elements *array* and *arrayref*, which are tied via location specifiers to array declarations and usages respectively. The constraint reads "If an array reference is made to array a with index i, and array a has been declared with lower bound l and upper bound u, l must be less than or equal to i and i must be less than or equal to u." The specification constrains the values of the variables. The TASPEC execution monitor flags an error condition if the constraint is violated.

Logic and temporal operations are used to group simple facts into more complex terms. Temporal operators allow relationships of predicates with respect to time to be expressed easily. Consider a special case of array bounds checking, determining whether character strings in C are null-terminated. The null-termination property can be expressed in TASPEC.

Two new predicates are defined:

code description	code	generated TASPEC state element
array element assignment	`a[i] = v`	$array\_contents(a,i,v)$
reference to whole array	`a`	$reference(a)$

The specification is that a string array has a null terminator if it has an element which is 0, i.e., if there is a match for $array\_contents(a,i,0)$ in the concrete state. The invariant that requires null-termination is:

$$reference(a) \wedge array(a,l,u) \Rightarrow array\_contents(a,i,0).$$

The *reference* predicate records a variable usage. The *array* predicate is checked to guarantee that the usage is to an array. The invariant says that if a reference is made to an array, there must be one element of the array with the value 0.

4.3.3 Location specifiers

Location specifiers are essential to tie TASPEC specifications to actual source code. Ideally, it should be possible through location specifiers to link specifications to any point in the program where the abstract state changes, depending upon the needs of the specification. Examples in this chapter limit locations to function calls, variable usages, variable declarations, and variable assignments. In general, there is a direct mapping between possible location specifiers and nodes in the data-flow graph of the program.

A conditional expression can be attached to a location; at run-time, the conditional expression is evaluated (if present) when the location is reached, to test if the location should match. If no conditional expression is present, the location always matches. For example,

$$\textbf{location func } \mathit{malloc}() \textbf{ returns } \mathit{ptr} \textbf{ if } \mathit{ptr} \neq 0 \; \{\ldots\}$$

matches all instances of the `malloc` library call that return with a non-zero value. All TASPEC objects within the {} activate if a match occurs.

A location specifier is compared against all potential matches in the program. All points in the program which match the location specifier are used. There is no mechanism for tying specifications to a specific point (i.e., a specific **malloc** call) in the program. This limitation is practical not theoretical; the interactive interface to the Tester's Assistant is limited. If, in the future, the Tester's Assistant becomes a more integrated testing environment, the ability to reference specific code points will be added.

Most of the location specifiers in specifications developed so far use functions as the location type. Functions, especially if the function is a library or system call, exhibit a simple narrow functionality, which makes for concise specification. Also, function specifications allow underlying details of the function body to be ignored. In the case of library and system calls this assumption is important; a coherent interface to the operating system can then be established.

For flaws that arise from interactions with the operating system, function location specifiers are sufficient, because system and library calls completely define the interface. Other flaws, such as array overflow, are caused by an incorrect variable access, so a variable usage location is required:

$$\textbf{location variable } a[i]\{\textbf{assert } \mathit{arrayref}(a, i);\}$$

The *arrayref* predicate is described in Section 4.2.1. In this specification, each location is a reference to an array element and the *arrayref* predicate is asserted.

Additionally, for the Tester's Assistant to be used for other applications (such as for safety), it is necessary to create specifications aimed at determining program correctness rather than the program's relationship with the security state of the operating system. Therefore, tracking the value of program variables is of more interest. Variable assignment and declaration location specifiers are of utility for this purpose.

Location specifiers are parameterized by copying data values from locations in the code. If a parameter appears in the location portion of a specification, it is bound to a value from the run-time state of the program when the location matches. Assertions, invariants, and pre-conditions attached to the location specifier use this bound value for the parameter. For example, in function location specifiers, the arguments to the function as well as the result value can be used as parameters. For variable based locations, array indices can be used.

4.3.4 Event processing

Event processing controls the construction of the concrete state, as well as the invariants against which the concrete state should be checked. Event processing is presented in two parts: first a description of the mechanisms for altering the concrete state, then a description of invariants and how the state altering mechanisms are activated with pre-conditions.

The state is changed by either asserting a concrete state element (adding it to the state) or retracting (removing) it from the state. An iterator can be used when setting up larger state constructs. The iterator is especially useful when initializing the concrete state with information about an array or similar structure. For example, in

$$\texttt{assert}(\mathit{index} = 0, n - 1)\ \mathit{array\_contents}(\mathit{array}, \mathit{index}, 0);$$

the iterator is used to initialize each element in an array's representation in the concrete state to 0. The assertion of *array_contents* is attached to static array allocation, which in C guarantees memory that is zeroed out.

location decl $array[n]\{$

$$\texttt{assert}(index = 0, n - 1)\ array\_contents(array, index, 0);\}$$

Note that this location provides values for the unbound parameters *array* and n in the assertion of *array_contents*.

The fundamental unit of specification is an event. An event can either be an assertion (actually either of the two methods for changing the state, **assert** or **retract**), an invariant, or a pre-condition. An assertion alters the concrete state. An invariant is a constraint against which the concrete state is checked. A pre-condition is an invariant that, if it matches the concrete state, causes other events to be activated. See Section 4.4 for a discussion on the evaluation model for processing the various events.

Consider the high-level property *null_terminated*(s), which signifies if the character array represented by its argument is terminated by the null character. This property is high-level because no single location specifier causes it to be added to the concrete state. In order for something to be null-terminated, it must be an array, and there must be an index into the array (within its array bounds) that contains the null character. The null-termination specification is expressed in TASPEC by using an invariant that activates the assertion of the predicate:

$$array(a, l, u) \wedge array\_contents(a, i, 0) \wedge l \leq i \wedge i \leq u\{\text{assert } null\_terminated(a);\}$$

4.4 Evaluation model and implementation issues

The current implementation of TASPEC in the Tester's Assistant does not have the complete capability to assess the correctness of an execution. In the Tester's Assistant at present, specifications are parsed and stored according to associated locations. Then source code for a program is instrumented so that when the program is executed, the basic asserts are emitted. The location specifiers currently used are function locations and variable locations.

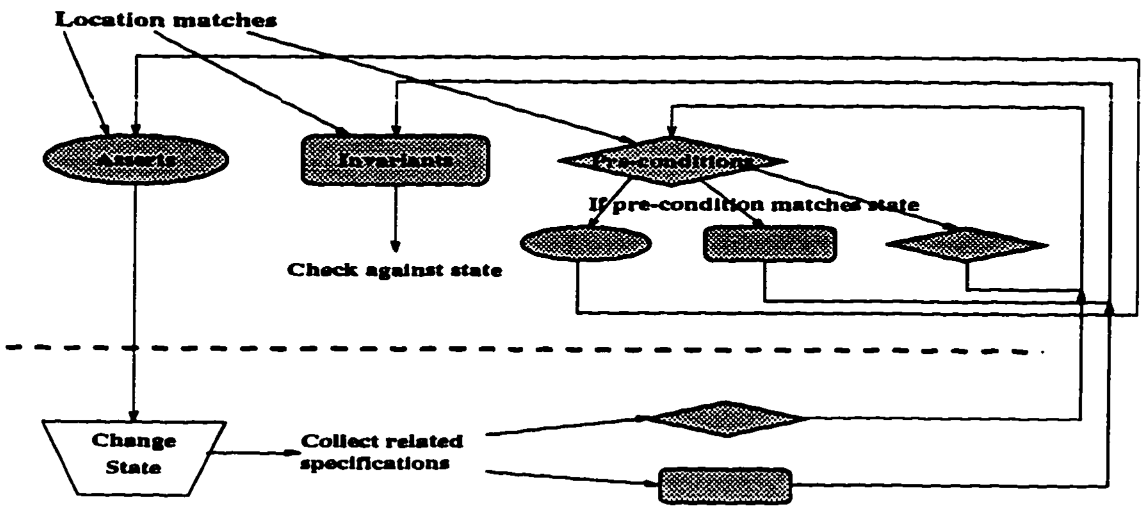
The actual correctness checker is still in the design stage. In the current design, location specifiers cause code to emit location identifiers and data values when they are executed. A separate process takes this information and evaluates it with respect to the specification invariants. Thus evaluating TASPEC serves as a test oracle for the executing program. A description of the evaluation model is in Figure 4.2. The remainder of this section describes the evaluation model in more detail.

A matching location activates one or more events, each of which is an assert, invariant, or pre-condition in the body of the location specifier. TASPEC unbound variables are bound to values of variables from the run-time state of the program. Any valid variable expression can be used to extract data. However, structure and array dereferencing must be used with care because a complex memory reference is directly translated to source code for use in execution monitoring, so references must be exact and accurate for execution monitoring to be successful. Binding of TASPEC variables also occurs when a pre-condition matches the concrete state.

Once values for bound variables are inserted into the activated events, each event type is processed in its own way. Asserts (and retracts) are stored in a pool of asserts to be activated after all of the invariants and pre-conditions have been checked against the current state. If a pre-condition matches the state, all asserts, invariants and pre-conditions which are in the body of the pre-condition's event need to be checked by the same process.

Once all pre-conditions have been checked, then all asserts and retracts that have been pooled are executed simultaneously. Processing asserts and retracts causes global invariants and pre-conditions (not attached to specific locations) that depend upon the predicates which have been altered to be re-evaluated. Re-evaluation is done by the same agent that resolved the original events raised by the location specifier.

The actual checking of invariants and pre-conditions can be done by a Prolog interpreter. Prolog is well-suited to checking a database of predicates based upon partial knowledge. Prolog can also backtrack in an effort to find multiple predicates that match with common data. Using the Prolog search engine minimizes the effort used to engineer the execution monitor.



Each shaded object is a set of one type of events: ovals are asserts and retracts, rectangles are invariants, and diamonds are pre-conditions. Arrows trace the flow of events during evaluation. The box in the lower left corner is the agent that actually alters the concrete state. As a result of the alteration, new pre-conditions and invariants need to be checked.

Figure 4.2: Evaluation model for TASPEC.

4.5 Completeness

For TASPEC to be a viable tool for testing and analyzing programs, it must be able to express any possible property of a program; i.e., TASPEC must be complete. There is no standard definition of completeness. The definition of completeness used in this section is:

> *A program state includes the values of all program variables and indicates what code is currently being executed. A property specification language is* complete *if it can represent relationships between program states at different points in the program's execution with arbitrary first order formulas.*

A program can be described by enumerating all of its internal states and completely recording its interaction with its external system. By examining what code is being executed, the interaction with the external system can be monitored. Therefore, the program state at different points in the program completely describes the program. A specification language which can express any portion of this description is complete. This section shows that TASPEC satisfies this definition.

The proof of completeness is shown by construction. First, a location specifier extracts program values. Second, an event attaches a time stamp to every other predicate in the specification. Third, predicates link with the basic constructors of first-order logic. These three constructions together show that TASPEC can represent the first-order formula required by the definition of completeness.

Attaching a location specifier to an variable assignment extracts the value of the variable after the assignment:

$$\textbf{location assign } a \textbf{ result } b \ \{\text{assert } value(a, b);\}$$

The second parameter of the *value* predicate keeps track of the latest value for a variable. This specification works for simple objects with no complex memory references. The specification of array bounds throughout this chapter and in Section 4.6.3 shows how this technique can be applied to more complex memory structures.

The *value* predicate can be further augmented by a predicate *time*. The *time* predicate contains an index indicating the number of locations that have been matched so far in the execution of the program[2]. To every event list attached to a location specifier in the specification, add the pre-condition and event list

```
location ... {
 time(t) {
     retract time(t);
     assert time(t + 1);
     }
}
```

Then, add a third parameter to the *value* predicate indicating the present value of time:

$$time(t) \ \{\text{assert } value(a, b, t);\}$$

[2]This implementation of time is specialized for the purposes of the proof. Simpler implementations of time are possible for individual specifications. A future implementation of TASPEC may support automatic time-stamping (see Section 4.7).

Now specifications about the value of a variable in time are possible, by adding relations on the time parameter. For instance, to say that the value of the bound variable a never decreases,

$$value(a,b,t) \wedge value(a,c,u) \wedge t \leq u \Rightarrow b \leq c.$$

The parameters b, t (and also c and u) are unbound, so they match all instances of $value(a,b,t)$ in the concrete state. Additionally, the TASPEC operators **before**, **until**, and **eventually** allow abstract temporal properties to be specified.

All parameter variables in a TASPEC specification can be quantified. Because TASPEC specification states are created in a bounded manner when a program executes, any quantification is done in a correspondingly bounded manner. By default, every unbound variable in a TASPEC predicate is universally quantified, and once a variable is bound in a predicate, it is bound throughout the formula containing the predicate. So, the formula

$$arrayref(a,i) \wedge array(a,l,u)$$

is interpreted as

$$\forall a,i\,(arrayref(a,i) \wedge \forall l,u\,(array(a,l,u))).$$

Remember that the universal quantification is over the concrete state, so this matches first all instances of *arrayref* and then all instances of *array* that correspond. Existential quantification is stickier; the second clause of the above formula fails if there are no instances of *array* with a first parameter of a, so this formula establishes the existence of such an *array* predicate with the given values. However, there is no way to distinguish a single set of values which satisfy the formula, so existential quantification is only partially supported. In classic first-order logic, $\neg\forall\neg$ is equivalent to the existential quantification $\exists$, but this construction is not possible in TASPEC because of the implicit universal quantification at each predicate. An extension to TASPEC which allows full existential quantification would be trivial (perhaps by adding a choose operator @ which is a prefix to a predicate), as it is just a special case of the universal pattern matching which is already present. The other necessary operators for first-order predicate logic, $\wedge$ and $\neg$ are explicitly provided by TASPEC.

Therefore, with the possible drawback of partial existential quantification, TASPEC has been shown to be complete.

4.6 More examples

This section contains complete examples of TASPEC specifications for the example programs introduced in Chapter 3. Each sub-section describes a generic flaw or property which has been at the root of Unix security flaws. A program in which the flaw was present is described in each section as well.

4.6.1 Authentication and ftpd

Programs that control access to multi-user systems must somehow ensure that the user who is granted access to a system has proper authority; in Unix systems this authority is gained by presenting the user's password which is on file in the system. The process of gaining authority is called authentication. A generic property for this class of programs is that authentication is correctly performed before access to the system is granted.

Ftpd

Ftpd is the server for ftp (File Transfer Protocol), a protocol which provides a file transfer utility between two computers connected by a network. Section 2.1 briefly describes the flaw present in some versions of ftpd flaw. In more detail, the ftpd server implements a command loop, where a command is received from the client, the command is performed, and then the loop is repeated. One of the commands is a "login" operation, where the server implements an authentication routine, so that access to files is restricted to operations allowed by the access control policy. This routine produces two variables which are used in other parts of the code: a `uid` variable which stores a unique integer identifying the current user of the system, and a `logged_in` variable, which is set to 1 after a successful authentication. In correct versions of the code, the value of `logged_in` is semantically associated with `uid`, and `logged_in` is reset when `uid` is altered. In the flawed version, `logged_in` is not reset when `uid` is altered; therefore `uid` is recorded as being authenticated without successfully traversing the login routine.

The authentication property

The property which this flaw violates is an abstract predicate that relates authentication to permissions:

$$\mathit{authenticated}(\mathit{uid})\ \textbf{before}\ \mathit{permissions\_granted}(\mathit{uid})$$

The property says that the user with user id *uid* must be correctly authenticated before access is given to the system for that user. Both *authenticated* and *permissions_granted* are properties that are defined in other specifications.

The definition of *permissions_granted* is more straight-forward, and so is presented first.

$$\textbf{location func}\ \mathit{setuid}(\mathit{uid})\ \textbf{result}\ 1\{\text{assert}\ \mathit{permissions\_granted}(\mathit{uid});\}$$

This specification associates permissions_granted with a specific system call via a function location specifier. The system call `setuid` in Unix changes the permissions of a process to those of user `uid`, provided that `setuid` returns successfully, i.e., returns `1`. When this circumstance occurs in the execution of a program, the predicate *permissions_granted*(*uid*) is put into the concrete state with the correct value substituted for *uid*. When the `location` specification is combined with the abstract predicate, the `setuid` calls in the source code are targeted for further analysis as discussed in Section 5.4.

The definition of *authenticated* is more complex; the Unix system does not supply a standard system or library call that implements an authentication routine. Fortunately, however, the method for Unix authentication is relatively fixed. The plain text password entered by the user is encrypted with the salt that appears in the password file; the result must be the same as the password entry in the password file. This relationship is described in terms of the TASPEC specifications in Figure 4.3.

location func *crypt(password, salt)* **result** *encryptpassword*{
  assert *password_entered(encryptpassword)*; }

location func *getpwnam(name)* **result** *pwent*{
  assert *user_password(name, pwent → pw_passwd, pwent → pw_uid)*; }

location func *strcmp(s1, s2)* **result** 0{
  assert *equal(s1, s2)*; }

password_entered(pwd1) $\wedge$ *user_password(name, pwd2, uid)* $\wedge$ *equal(pwd1, pwd2)*{
  assert *authenticated(uid)*; }

Figure 4.3: Property specification for authentication.

The *authenticated* property is only instantiated in the concrete state if the three primitive predicates *password_entered*, *user_password*, and *equal* are in the state with the correct parameter values. The *password_entered* predicate records the return value for crypt in the concrete state.

The pwd library provides a C language interface to the password file. The getpwnam function retrieves all information about a specific user from the password file and stores it in a structure. The *user_password* predicate associates the password field of a user with the user's user id. This information is put into the concrete state at the time it is retrieved from the password file.

The final step in the authentication is a comparison of the encrypted form of the password obtained from the user and the encrypted form of the password from the password file. The *equal* predicate represents the fact that this comparison was made successfully.

The specification that asserts the *authenticated* predicate is an example of event-based processing; when the three predicates representing the pre-condition match the concrete state, the assertion in the body of the specification is made.

4.6.2 Race conditions and rdist

A race condition occurs when two processes compete ("race") to access the same data object. This section discusses race conditions arising from Unix file system operations. In Unix, a number of file operation functions take as an argument a file name rather than a file descriptor. A file descriptor binds file operations to an actual file, so that subsequent uses of the descriptor refer to the same file, even though the file may have changed names. If a process performs two consecutive file operations, using the file name as a reference, there is no guarantee that the second operation is performed on the same file. A second process could change the file referenced by the file name, resulting in a race between the second process and the second operation of the first process. If the second process actually succeeds in changing the file referenced by the first process, then the race condition has occurred..

The specification presented in this section does not attempt to express race conditions, but rather the width of the window that can be exploited by a second process. Often, the amount of time between the first file operation and the second operation is very small, so the probability that a second process could interpose an operation is also very small; this probability is dependent upon the operating system scheduler. However, if there is an operation which can block that comes between the first and second file access, and a user can control the item at which the operation blocks, then a program can be written which exploits the race condition 100% of the time. The specification presented in this section outlines a method for identifying and characterizing such blocking activities, thus detecting easily exploitable race conditions. With the specification, a single process can be examined for race condition vulnerability. Testing can be done independently of any potential second process; an error is detected if the circumstances arise in the main process where a second process could easily interfere.

Rdist

Rdist is a Unix program for maintaining file systems on distributed hosts. Rdist is described in detail in Section 3.1.2.

```
master                          slave
        loop: write()                   creat()
                                        loop: read()
                                        close()
                                        chown()
```

Figure 4.4: Pseudo-code for the rdist race condition.

Pseudo code of the rdist race condition is presented in Figure 4.4. If the master rdist is sending a file that has not yet been fully written, it blocks until the file is completed; thus the slave rdist also blocks on read. Therefore, a user executing rdist can force a blocking read, which allows a second process to change the file referred to by the file name while the rdist process is still blocked. Consequently, exploitation of the flaw is not dependent upon the scheduler executing the second process at the correct time.

Race condition specification

The specification in Figure 4.5 detects the case where a function call (e.g. `read`) could potentially block. Thus the race condition window between `creat` and `chown` is widened, making it easily exploitable. This condition is detected by the invariant attached to the `chown` location. If a *blocking* predicate with a parameter representing the file accessed by the `chown` is in the concrete state at the time of the `chown`, an error condition is raised.

The *blocking* predicate is asserted when executing an operation such as `read` which has the potential to block during execution. The *blocking* element is associated with a particular file, because if the file's permissions are re-checked by a function like `stat` then *blocking* predicates associated with the file checked by `stat` are removed. In this case, the window of opportunity narrows because up to the point of the `stat`, the file has not been altered by some other process.

The *accessing* predicate is asserted at the beginning of every window, in this case by the `creat` call. When the `read` call is performed, the pre-condition $accessing(File)$

```
location func read() {
    accessing(File){assert blocking(File); }
    }

location func creat(File){
    assert accessing(File);
    retract blocking(File);
    }

location func chown(File){
    ¬blocking(File);
    }

location func stat(File){
    retract blocking(File);
    }
```

Figure 4.5: Partial specification of the race condition window exploited in rdist.

is matched against the concrete state, and the attached event which asserts *blocking*(*File*) is activated. *blocking*(*File*) is asserted for every *accessing*(*File*) that exists in the concrete state.

4.6.3 Array overflow in C

Standard C compilers do not check array bounds at compile time, and the code they produce does not check array bounds at execution time. In most programs, all array references should be within the declared bounds of the array.

Dynamically allocated arrays and the use of pointer arithmetic to access elements of any array complicate the issue of array bounds. With dynamically allocated arrays, the size of the array is not known at compile time. Whenever non-standard array references are made (`a[i]` being standard), it may be unclear which array is being

location decl $a[n]$ {**assert** $array(a, 0, n-1);$ }

location variable $a[i]$ {**assert** $arrayref(a, i);$ }

location assign a **result** b{
    $array(a, l, v)${**assert** $array(b, l, u);$ }
    }

$arrayref(a, i) \wedge array(a, l, u) \Rightarrow l \leq i \wedge i \leq u$

Figure 4.6: Specification of static array bounds checking.

referenced. This section presents a simple set of specifications which detect array bounds violations in statically declared arrays, and a fragment of the extension to that specification for dynamically declared arrays.

Fingerd

The flaw in fingerd exploited by the Internet worm [59] was an array overflow error. Fingerd and its flaw are described in Section 3.1.2.

Specifying array bounds in TASPEC

If arrays are declared statically, the specification (in Figure 4.6) is quite simple. The *array* predicate corresponds with the declaration of an array. It contains upper and lower bound information as well as an identifier for the array as parameters. *Array* is asserted upon declaration and array information is transferred by means of an array assignment. If an assigned value has an *array* predicate, then a new *array* predicate is put into the concrete state.

The *arrayref* predicate is added to the concrete state whenever a reference to an array element occurs. The predicate records the array accessed and the index within the array which was accessed. An invariant specifies that the *index* of the *arrayref* must be between the lower and upper bounds of the array as recorded in the *array* predicate.

The static array bounds checking has problems with arrays declared in local scopes. When the local scope terminates, a locally-declared static array no longer exists, but concrete state elements referring to the array still exist. No location specifier matches the end of a scope, so it is impossible to specify that the *array* predicate is retracted. A new location specifier **undecl** may be necessary to specify events to occur when a variable becomes undefined.

Specification of array bounds checking for dynamically allocated arrays is an extension of the static allocation specification. For clarity, new predicates have been defined, *object* and *objectref* which correspond semantically to *array* and *arrayref* respectively. An additional parameter to *object* denotes the size (in bytes) of an individual array entry. The invariant is virtually identical to the invariant in the simpler case. The specification presented in Figure 4.7 introduces the major elements of the specification; more locations are necessary to completely handle all pointer transactions.

The difference between the dynamic and static specifications is the *objectptr* predicate. With this predicate, a distinction is made between the declaration of memory (through dynamic allocation) and a pointer to the declared object; a pointer does not necessarily point to the beginning of an object (e.g., if it has been incremented). The parameters of *objectptr* correspond to the pointer, the block which the pointer references, and the index into the block. *Objectptr* predicates are kept up-to-date by the declaration and assignment specification, and they are used in the assertion of *objectref* predicates.

The `calloc` call creates a new memory object, therefore new *object* and *objectptr* predicates are also created. In the dereferencing location ($*a$), an *objectref* predicate is generated based upon the *objectptr* currently in the state for the pointer. In the assignment location, either the right hand side of the assignment corresponds to an *objectptr* predicate already in the state, or there is not an *objectptr* already in the state, in which case the right hand side has computed an address to assign to the pointer variable. The first case is simple; the *objectptr* in the state corresponding to the left hand side is altered to reflect the pointer's new meaning.

When pointer arithmetic computes a pointer variable, the specification must determine what the meaning of the new pointer variable is. The assumption in the specification is that if pointer arithmetic is being performed, then the resulting pointer must point within the memory object that the left hand side had previously referenced. The memory object that a pointer referenced is recorded with the $objectptr(b, d, j)$ invariant. With the $objectptr(b, d, j)$ and the *object* predicate for d as pre-condition, the new value of $objectptr(b, \ldots)$ can be asserted. The index is computed based upon the offset from the base of the array, using the size of an individual element in the arithmetic computation.

When an object is de-allocated with `free`, all *object* and *objectptr* predicates corresponding with the object are removed from the concrete state.

4.7 Conclusions

TASPEC is a good language to express properties because of its simple evaluation model and because of its ability to correlate events with code through location specifiers. Test oracles are automatically generated from TASPEC specifications, and there is potential for relationships between specifications and other aspects of testing such as coverage.

TASPEC's approach in associating specifications with code is similar to that of Larch [24], but the concept of a TASPEC location is more general than Larch's function-level specifications. Many established languages such as Larch and Z [61] have tools and methodologies to allow formal reasoning about specifications. Currently, TASPEC has no formal reasoning facilities, so the consistency and correctness of TASPEC specifications cannot be guaranteed.

Further development of the TASPEC language is necessary. No distinction is made in the syntax between an identifier and a variable. For example, it should be possible to have a location match a reference to a specific array, and also possible to have another location match a reference to any array. Additionally, specific program locations cannot be referenced by program specifiers. It should be possible to have a location match one specific array reference in the program. As more of the

```
location func $calloc(n, s)$ result $addr$ {
    assert $object(addr, 0, n-1, s)$;
    assert $objectptr(addr, addr, 0)$;
    }

location variable $*a${
    $objectptr(a, c, i)$ {assert $objectref(c, i)$; }
    }

location assign $b$ result $a${
    $objectptr(a, c, i)$ {
        assert $objectptr(b, c, i)$;
        retract $objectptr(b, d, j)$;
        }
    $\neg objectptr(a, c, i)$ {
        $objectptr(b, d, j)$;
        $objectptr(b, d, j) \wedge object(d, l, u, s)${
            retract $objectptr(b, d, j)$;
            assert $objectptr(b, d, (a-d)/s)$;
            }
        }
    }

$objectref(a, i) \wedge object(a, l, u, s) \Rightarrow l \leq i \wedge i \leq u$

location func $free(a)${
    $object(a, l, u, s)$ {retract $object(a, l, u, s)$; }
    $objectptr(a, b, i)$ {retract $objectptr(a, b, i)$; }
    }
```

Figure 4.7: Array bounds checking with calloc.

implementation of TASPEC is accomplished, more language issues could arise.

Chapter 5

Static data-flow analysis

A data-flow graph is a directed graph that describes the flow of information in a program. Information flow is identified by tracking dependences between program statements and variables. For example, with a data-flow graph it is possible to make assertions like *the value returned from this call to* `read` *affects the behavior of these other parts of the program.* This assertion describes information flow from the `read` statement.

Every object and operation in a program is represented by a node in the data-flow graph. Edges in the data-flow graph represent a variety of types of information flow between nodes. For example, if a value is used in an assignment, the graph has a "data" edge between the node representing the value and the node representing the result of the assignment. If the assignment is in the `else` branch of an `if` statement, then there is a "control" edge between the assignment node and the node representing the `if` because the value of the `if`'s predicate influences whether or not the assignment is executed. The most important information flow to calculate is the flow from the definition of a value in an assignment statement to the use of that value in an expression elsewhere in the program. This flow is represented by a "def-use" edge, short for definition-use. Def-use edges are also the most difficult to calculate, because they are not directly derivable from program structure, as is the case with data and control edges. Def-use edges must be calculated by "propagating" definitions through the program, to determine the uses that each definition reaches.

The information flow stored in a data-flow graph is important for the validation of properties of the program that the graph represents. A property is a constraint upon values of variables in the program. Assignments to the variables are represented as nodes in the data-flow graph. The important information for the validation of the property is the computation which results in the final value of the assignment, and also the input values on which the value of the assignment depend; i.e., the slice of the program with respect to the property. This information is directly available from the edges of the data-flow graph.

Data-flow graphs are typically computed with control flow graphs (CFG) and program dependence graphs [3, 20, 36, 43, 51, 52, 63]. CFG-based algorithms produce a program-independent CFG before propagating definitions. Static analysis in the Tester's Assistant is novel primarily because def-use edges are generated by propagating definitions through the structure of the program, without producing an explicit CFG.

Portions of Tester's Assistant static analysis algorithms are similar to, or are derived from, related work. The concept of basing static analysis on fine-grained representation of the program rather than a statement-based or CFG-based representation was developed in parallel with [15]. The algorithm for analyzing unstructured control constructs was developed in parallel with [3]. The algorithm of [43] is used to analyze recursive functions. Personal experience suggests that most static analysis techniques are similar to each other, but no benchmarks exist for comparing static analysis algorithms [28].

This chapter describes data-flow analysis and the specific algorithm used in the Tester's Assistant. Section 5.1 is an overview of the issues and problems in constructing a global data-flow graph of a C program statically. Section 5.2 describes the exact structure of the graph. Section 5.3 gives a more precise description of the construction algorithm. Section 5.4 describes the use of the data-flow graph in slicing and other areas of the Tester's Assistant.

5.1 Overview

Data-flow analysis for complex languages like C is very difficult. These languages contain a number of features and constructs which are very difficult to analyze, including:

- Global variables. A function can cause side effects on any number of global variables; keeping track of what global variables are possibly affected in an entire sub-tree of the call graph can be difficult.

- Pointer aliasing. Through use of pointer assignment, two pointer variables can point to the same object in memory. Then, an assignment to one of the (dereferenced) pointers actually affects the value which the other pointer references. Keeping track of what pointers could possibly be aliased to each other is very difficult, especially when the pointers refer to dynamically allocated memory objects.

- Pointer arithmetic. In C, it is possible for two pointer variables to become aliased through pointer arithmetic rather than by mere assignment. A pointer which is continually incremented will eventually point to a portion of memory outside of the object that the pointer originally referenced. Keeping track statically of what memory object a pointer could reference is impossible, in general. This problem is the reason why complete static analysis of C is impossible in general.

- Type-casting and union structures. A union structure allows objects which represent two different semantic entities to share memory space. This presents another source of aliases: an assignment to one of the objects is also an assignment to the other object, because of the shared memory. Type-casting a pointer variable has the similar effect of aliasing the objects referenced by the pointer before and after the type-cast.

- Gotos. A `goto` is an unstructured control construct. With structured control constructs such as `if` and `while`, jumps are made in a regular fashion, and so are easy to analyze. A goto affects the flow of control of the program in an unstructured way, in that the target of the goto can be anywhere else in the function; the jump can be across, into, or out of any number of structured control constructs. When a goto is involved, it is very hard for usage of a value in an expression to be associated with the possible assignments which may have produced the value. [3]

These problems make accurate static analysis of C programs difficult, even for those programs for which it is possible to complete static analysis. Thus, it is necessary to make some approximations and reduce the accuracy or thoroughness of the analysis in order to increase the tractability of the problem. When making decisions on which heuristics to apply, it is necessary to define what information is or is not important to the end product, and where lack of accuracy does not negatively impact the result. For example, finding a def-use dependency that actually does not exist does not result in a false positive during testing; if the additional def-use dependency is required, it can never be covered. One common goal of a data-flow analysis heuristic is "safety." An analysis is safe if it does not miss any information flow which actually exists in the program. The analysis may be inaccurate in that it might record some false dependences, but no actual dependences may be absent.

A possible assumption is that code that is too complex to be analyzed is too poorly written to be trusted. The fact that the code is too complex is a meaningful result for the testing process. Code of this nature, once identified, can either be hand-analyzed, or marked un-validated.

In the Tester's Assistant, the purpose of static analysis is to produce and evaluate tests. Less accurate static analysis is tolerable if significant speed-ups occur as a result. However, reduced safety is not tolerable, because of the goal of property validation; a false negative result for a property is acceptable, a false positive is not acceptable. Static analysis produces inaccurate data-flow graphs if the program is not *well-behaved*, so independent testing of well-behavedness is necessary prior to static analysis. Section 5.1.1 discusses well-behavedness in more detail.

The Tester's Assistant is not intended as the whole testing process; other tools must be added. Some analyses are better performed by other tools. In particular, the syntax checking programs such as "lint" do a variety of structural checks on C programs, and reports any errors or type violations. Due to the presence of these tools it is possible to assume that code which is analyzed by the Tester's Assistant is free of the sorts of faults that lint detects. The Tester's Assistant static analysis is un-safe when viewed in isolation, because errors of the type found by syntax checkers could cause Tester's Assistant to produce an un-safe static analysis; with the addition of "lint" and a well-behavedness checker, however, safety is restored.

Finally, pointer aliasing and type-casting problems are not dealt with in the current implementation of the Tester's Assistant. Preliminary designs have been made on how to calculate, store, and utilize alias maps which can handle these two problems, but solving the problem remains future work.

As stated above, the major part of static analysis is finding the edges from nodes representing assignments to nodes representing uses of the values generated at the assignments. In the Tester's Assistant, this step is called "live assignment propagation." An assignment to a variable is live at a node in a program if in some possible execution of the program the current value of the variable was determined at the assignment. Live assignments are propagated along possible paths of execution; each node stores a list of assignments which are currently live.

Because static analysis must take into account every possible execution, live assignment lists contain every assignment node that could, via some execution path, be currently live. Therefore, a live assignment list can contain more than one assignment to the same variable.

5.1.1 Well-behavedness

Static analysis depends heavily upon the correctness of data-flow analysis for the target program. In this section, we discuss the well-behavedness problem, which arises from pointers referencing unanticipated memory locations. In the Tester's Assistant, programs are assumed to be well-behaved in order to simplify static analysis. Other

```
a = 1;
b = &a;
*b = 2;
printf(a);
```

(a)

```
int f(int *i,int *j)
{
  *i = 1;
  printf(*j)
}
f(&x,&x);
```

(b)

```
int a[10];
for(i = 1;i < 11;
i++)
    a[i] = 0;
```

(c)

(a) pointer variable `b` points to `a`, so an assignment to `*b` affects the `printf`.

(b) The behavior of the function `f` is changed when it is called with its two arguments aliased to each other.

(c) This code fragment is not well-behaved because `a` is of size 10, but an assignment is made to the 11th element.

Figure 5.1: Three examples of potentially ill-behaved code.

tools such as lint or [45, 44] can be used to establish well-behavedness. See Figure 5.1 for examples of well-behavedness problems.

If an external tool is not available, property-based testing also can be used to ascertain a property such as well-behavedness. Assume that the ways in which programs can be ill-behaved can be described, and use property-based testing to validate these. Let P be the first point in the program where it starts to act ill-behaved. Then the data-flow analysis up to P is correct, and so property-based testing is successful at finding the flaw at P. Once that flaw has been found and corrected, the next flaw can be addressed. However, any testing up to this point must be redone because ill-behavedness can invalidate the data-flow analysis upon which the testing was based.

The final point to make about well-behavedness is that for extremely obfuscated code, it may be difficult to determine if the code is well-behaved or not. In this case, the tester can label the source code "suspicious", based on its incomprehensibility. To a certain extent this begs the question. The Tester's Assistant is designed to aid a human tester in testing programs which are too large and complex to thoroughly test otherwise; it weakens the result if code which is too complex cannot be handled. However, when a secure system is the goal, saying that a program is too complex to be analyzed is almost as valuable as saying the program is broken. The portions of a program which are too complex should be rewritten so that the whole program is testable, even if the rewrite affects performance. High performance is less important than safety and security.

5.2 Internal program representation

This section describes the structure of the data-flow graph, describes the construction of node and edge types of the graph, and presents an example data-flow graph for a small function.

The Tester's Assistant data-flow graph has the following node types:

- Function declaration (`FDecl`). This node represents the function definition. It stores function-wide information such as the list of globals referenced within the function and the list of jump targets within the function.

- Function call (`FExpr`). This node represents a call site. It stores a reference to all the different parameter nodes of the function represented by the call site.

- Parameter node (`Para`). This node represents a parameter at a function call site. It records the side effect status of the parameter during the function call, and serves as an assignment node if the parameter is side effected.

- Control node (`CStmt`). This node represents a structured control construct (`if`, `while`, etc...) in live assignment propagation.

- Assignment node (`Asgn`). This node represents an assignment to a variable.

- Usage node (`IdUse`). This node represents the usage of a defined value.

- Declaration node (`Decl`). This node represents a variable declaration. It is not used in data-flow analysis but is useful in reconstructing executable programs from data-flow graphs.

- Jump node (`JStmt`). This node represents an unstructured branch, such as **`goto`** or **`break`**.

- Label node (`LStmt`). This node represents the target of an unstructured branch. A **`case`** statement is another example of an LStmt.

The first three types are necessary structurally to represent multiple functions and interprocedural slicing. The middle three represent the standard node types in a data-flow graph. The declaration node is a convenience node, so that we can find declarations which are useful when we output a subset of the original C code. The final two nodes are basically control nodes, but we separate them from structured control nodes in order to more directly address the complications in the live assignment propagation algorithm that such nodes cause.

Each node stores a constant amount of basic information (such as what part of the original program it corresponds to), and also stores all edges for which the node is an endpoint. Additionally, it stores all live assignments at that point in the program.

The graph has the following edge types:

- Declaration edge; an edge from a node representing a variable to the declaration of that variable. Like a Decl node, this edge is a convenience used when getting a valid C program from the data-flow graph.

- Control dependency edge; an edge between control nodes and the nodes on which they directly depend, i.e., nodes whose execution depends directly upon the result of the predicate represented by the control node. Each node has at least one control dependency edge attached to it, linking it with the CStmt or the FDecl which begin the block containing the node. If gotos are present, a block can have more than one entry point, in which case additional control dependencies are made to the additional entry points.

- Data dependency edge; an edge between a node which uses the result of a computation and a usage node which is part of that computation; for example, nodes on the right hand side of an assignment or in the predicate of a control structure. A usage node has at least one data dependency edge, but could have more than one if more than one computation is involved in the same usage; e.g. in `if (x++) ...` the usage node for `x` is used both in the computation of the assignment to `x` and also in the computation of the branch predicate, and so would have two data dependencies.

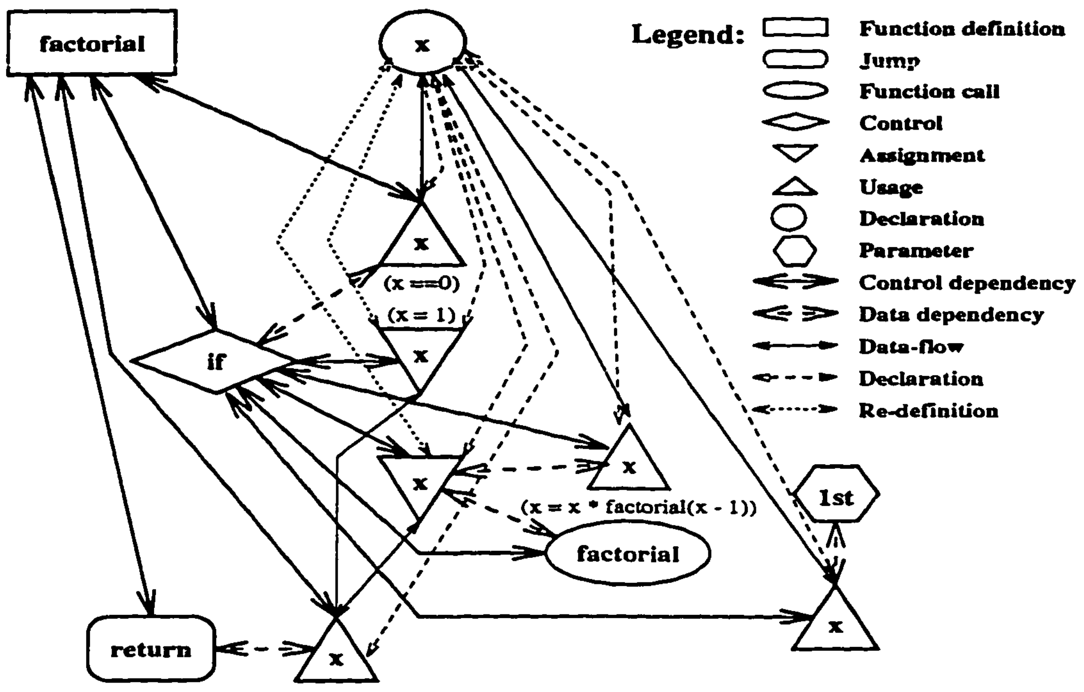
- Data-flow (def-use) edge; an edge between a usage of a variable and a definition of that variable. There is always at least one data-flow edge for each usage node. If there is more than one possible definition of a variable, there is one data-flow edge for each possible definition. For example, if each side of an "if-else" has a different definition of a variable, then there are two data-flow edges to an ensuing usage of the variable, one from each definition.

- Re-definition edge; an edge between a definition of a variable and a node that potentially redefines it. This is the only edge type which does not represent information flow in the program represented by the graph. A re-definition edge is used in calculating coverage contexts.

All of the edges except for the last two are calculated strictly from the structure of the program; the last two require the more complex live assignment propagation to calculate.

All of the edges except the re-definition edge represent information flow in the program the graph represents. To illustrate, assume that some node is of particular interest. The computation which determines the behavior of the node during any execution of the program is found by calculating the sub-graph containing all nodes from which information flows to the target node. Finding the sub-graph can be done by simply finding the transitive closure in the graph over the operation of traversing all the information flow edges backwards, i.e., from an IdUse to an Asgn, from the Asgn to the CStmt, and so forth.

Figure 5.2 is an example data-flow graph of the factorial function. Note that there are no edges in the graph from either a function definition or call site to parameters, or to return values. Information about a function's data-flow graph is stored in a special



```
int factorial(x)
int x;
{
  if (x == 0) x = 1;
  else x = x * factorial(x - 1);
  return x;
}
```

The top-level statements and expressions in the function are all control-dependent on the function definition node (solid bold edges), because these nodes are executed only if the function is executed. In turn, the nodes on both sides of the `if` are control dependent upon the `if`. The incoming parameter of factorial (the circled x) serves as the declaration and the initial value of the variable. There are data-flow edges between this node and all the uses of the variable x in the function, with the exception of the use of x in the return value, because the value of x at the return value is a result of one of the two assignments on the branches of the `if`. The bold dotted edges are data-dependency edges. The `if` is dependent upon x because x appears in its predicate. Likewise, x is used in an assignment and is used to determine the value of the parameter to the recursive call of factorial.

Figure 5.2: The data-flow graph for the factorial function.

function summary where all the information about a function's data-flow signature are stored, so that this information can be applied at each call site of the function (see Section 3.3 for more information).

A function summary has the following information for each of its arguments and its return value:

- A flag which indicates whether the parameter is an input or output parameter, or both
- A list of input parameters on which the output parameter depends
- A list of assignments inside the function which assign a result value to the output parameter

In addition, the function summary keeps a link to every return statement in the function, and links to all the call-sites.

Global variables referenced within a function are treated as virtual parameters to the function. For the purposes of the data-flow analysis, an extra call-by-reference parameter for the global is added to each function in which the global is referenced. Therefore, for each global variable referenced, the parameter information needs to be maintained. For a program containing many global variable references and many nested functions, this bookkeeping can be quite expensive. An additional factor in the cost of this bookkeeping is that a `Para` node is needed for each virtual parameter of a function at every call site of the function. So the use of a single global variable inside of a function causes an extra node to be created at every call site of the function.

This case is the only exception to the general rule that the number of nodes in the data-flow graph is directly proportional to the size of the program. The number of edges grows at a slightly larger rate, because most of the edge types can be connected to the same node more than once, and with virtual parameters a great many extra edges are necessary, especially within deeply nested function calls.

Figure 5.3 shows the size of the data-flow graph for a few programs. The five programs are taken from Ultrix, a variety of Unix. Ftp has been described above. The functions of the remaining four programs are: *Lpr* prepares a file to be printed,

	Lines of Code	Parse Tree Size	Number of nodes	Number of edges
lpr	1334	15232	3219	6472
ftp	5163	64694	13397	32731
ypserv	1972	31238	4814	7966
rwhod	535	8559	1255	3840
at	930	8294	1460	3226

Figure 5.3: Size of the data-flow graph compared to lines of code and size of the parse tree.

ypserv handles remote database requests, *rwhod* handles remote requests to identify the active users of a system, and *at* sets up a compute job to run at a later time.

5.3 Construction of the data-flow graph

This section describes the algorithms used to construct the data-flow graph of an arbitrary C program. The main part of the algorithm propagates live assignments throughout the data-flow graph; from this propagation, data-flow and re-definition edges are calculated. The other edge types and most of the edges are created in a pre-processing step based upon a parse tree of the program.

5.3.1 Pre-processing

The basics of pre-processing are presented in Figure 5.4. First, the ELI language translator toolkit [22] and the C language description which comes with the toolkit create a concrete syntax tree, which is abstracted to produce an abstract syntax tree (AST) that serves as the basis for semantic analysis. The AST has over 50 node types, and has almost 200 different rules associated with it. Figure 5.5 is an example AST.

The second step is to create the nodes for the data-flow graph, which represent the shaded nodes in Figure 5.5. Data-flow graph nodes are explicitly associated with nodes in the AST for the purpose of re-constructing portions of the original program. The AST contains information that allows it to be formatted with correct C syntax;

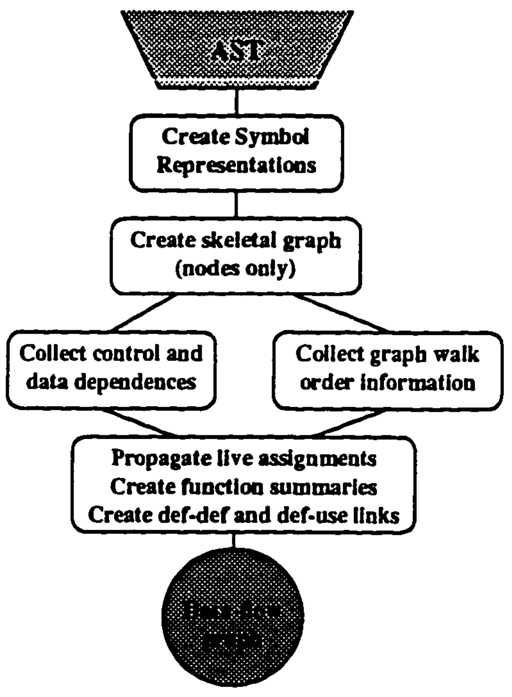

AST
Create Symbol Representations
Create skeletal graph (nodes only)
Collect control and data dependences
Collect graph walk order information
Propagate live assignments
Create function summaries
Create def-def and def-use links

Figure 5.4: Broad outline of the pre-processing step.

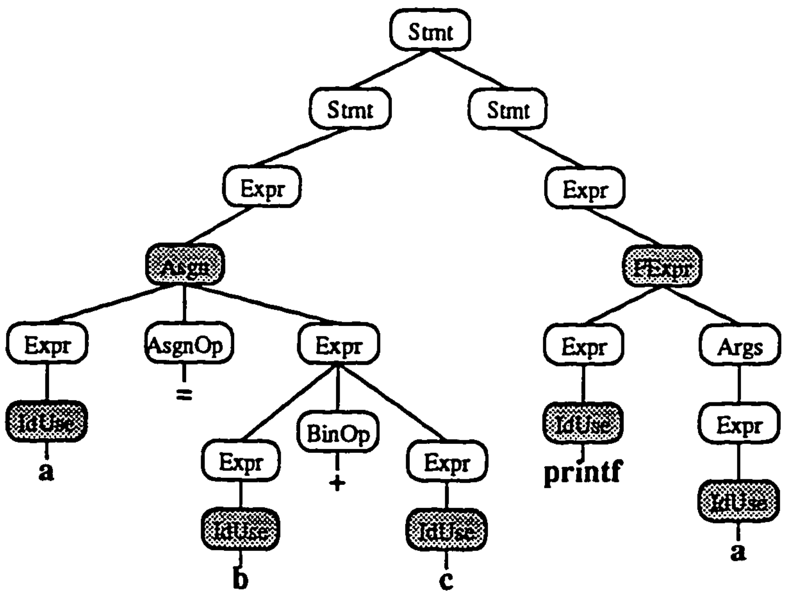


Figure 5.5: AST for the program fragment `a = b + c; printf(a);`. Shading represents nodes which are also in the data-flow graph.

links from data-flow graph nodes to AST nodes allow the data-flow graph to displayed with the same formatting information.

Once the nodes are established, edges can be created. Edges belonging to most edge types can be created from a single pass through the AST. These edges are called "static" edges. For example, assignment nodes are *data-dependent* upon usage nodes (data-flow nodes linked to `IdUse` nodes) on the right side of the assignment statement. Data-dependency can be determined statically by finding all usage nodes in the AST sub-tree on the right hand side of the `Asgn` node.

Control dependency edges are calculated by looking at subtrees of CStmt nodes; all nodes within a subtree are control dependent upon the CStmt except nodes which are "blocked" by other CStmt nodes. For example, in the program fragment

```
if (x)
  if (y) return 1;
```

the JStmt node representing the `return` has a control dependent edge with `if (y)` but not `if (x)`. The CStmt node representing `if (y)` has a control dependency edge with the `if (x)`.

The other main piece of information which can be easily obtained from the AST is execution order. The nodes which have been flagged as part of the data-flow graph are organized in hierarchical lists which denote evaluation order. These lists are constructed by walking through the AST and collecting nodes in the appropriate order. A list is a single sequential series of nodes. Some nodes, like nodes representing branching constructs (if, while, etc.) have sub-lists which contain the nodes on the various branches of the construct. The node execution order lists are used by the live assignment propagation function.

5.3.2 Live assignment propagation

The live assignment propagation algorithm symbolically evaluates the program to calculate the assignments that are possibly live at each node in the program. Because live assignments are the only information of interest for dependency analysis, further information regarding the possible values of the variables is discarded.

```
                                               LV = {}
(1)   x = read();                              LV = {1}
(2)   y = read();                              LV = {1,2}
(3)   if (y == 1) {
(4)        x = 1;                              LV = {2,4}
(5)     }                                      LV = {1,2,4}
(6)   if (y >= 10) {
(7)        x = y % 10;                         LV = {2,7}
(8)        y += 5;                             LV = {7,8}
(9)     } else {
(10)       x = y * y * 2;                      LV = {2,10}
(11)       y += 10;                            LV = {10,11}
(12)    }                                      LV = {7,8,10,11}
```

The current live assignment list after every statement is shown as auxiliary variable LV.

Figure 5.6: Propagation of live assignments.

For sequential code, the algorithm is trivial; simply transfer a list of live assignments from statement to statement sequentially. If a statement is an assignment or has side effects, then change the list to reflect the new mapping of variables to assignments. For code with branches, the assignment list on each branch represents a possible mapping of variables to assignments for that branch; so merge together the lists for the two branches.

The list of assignments which could be currently active, or "live", is maintained at every node. There is possible duplication of assignments on this list; two assignments to the same variable could be on different branches of an `if` statement, they both could be active after the if, so both assignments appear on the live assignments list for subsequent nodes.

To illustrate, consider the program in Figure 5.6. Live assignments (the auxiliary variable LV) are propagated throughout the program. The lists are considered separately for each branch; the `if` side of the if statement at (6) has a separate list than the `else` side. To determine the list at the end of the whole if statement, the lists from each branch are merged into a single list (without duplicates – the assignment

(2) still only appears once at the end of the first `if` statement).

This representation suffers some loss of accuracy. In Figure 5.6, the final LV list {7,8,10,11} suggests that there are four possibilities for the state corresponding with the two possible assignments for each of the two variables. However, there are only really two distinct states; the state in which x is assigned at 7 and y at 11 is not possible in any execution.

Looping constructs further complicate the calculation of live assignments. An assignment that appears later in a block can be used earlier in the block in a later iteration of a loop. Adjustments must be made to the algorithm to address this complication. There are two issues: first, getting correct live assignment lists for statement within the loop; second, calculating the live assignment list to propagate to the statement that is the successor to the looping statement.

In order to resolve in-loop live assignment lists, the propagation step is repeated for the nodes in the interior of the loop. Because live assignments that propagated to the end of the loop in the first pass are live for the second iteration of the loop, the ending live assignment list is copied to the beginning of the loop before the second propagation step. The live assignment list that passes to the statement after the loop merely needs to be the list after the first pass, possibly merged with the list after zero passes if the loop predicate is evaluated before the loop is executed once (and therefore the body of the loop is possibly never executed). An informal proof of this solution follows.

For the in-loop half of the solution, the solution is not sufficient if an assignment has not been propagated to a variable use in the loop when an actual execution could use the value from that assignment. If the assignment is before the start of the loop, or is in the loop and is before the use, then it would be propagated in the first pass through the loop, because of the reliability of the sequential portion of the algorithm. If it is after the use, and the assignment reaches the end of the loop, then it will be propagated during the second pass, because the assignment is included in the initial live assignment list for the second pass. The remaining case is that if the assignment does not reach the end of the loop, but in this case the assignment could not reach the use.

Construct	List passed to successor
if (e) S;	LV + e || LV + e + S
if (e) S1; else S2;	LV + e + S1 || LV + e + s2
while(e) S;	LV + e || LV + e + S + e
do S while e;	LV + S + e
for(e1;e2;e3) S;	LV + e1 + e2 || LV + e1 + e2 + S + e3 + e2

+ indicates the processing of live assignments in sequence. || indicates the merge operator. LV is the live assignment list at the beginning of the construct.

Figure 5.7: Live assignment propagation across structured control constructs.

For the successor half, the solution is not sufficient if there is an assignment propagated in error, or there is an assignment that is not propagated when it should be. No assignment is propagated in error; an assignment is propagated if it has propagated to the end of the loop on the first pass. The case where the loop executes once is a legitimate execution of the program, so there is some execution where the assignment reaches the end of the looping construct. Similarly, an assignment which is not propagated when it should be does so because some other assignment follows it and supersedes it by assiging to the same variable. However, this violates the assumption that assignments are propagated correctly in the sequential case. This solution for live assignment propagation across loops results in the live assignment equations for the various control constructs shown in Figure 5.7.

Adding unstructured control constructs complicates live assignment propagation because of the complexity of the graph traversal and the indefinite termination condition. The problem is that when the control flow graph contains loops, variable definitions which occur late in the function can be used earlier in the function. When the loops are structured (i.e., with while, for), two iterations of the linking function between basic blocks are sufficient to propagate all the definitions.

However, when loops are unstructured, like a `goto` (and thus may not be properly nested), the solution above that each loop be analyzed with a single additional propagation pass becomes complex, as what nodes actually compose the loop is ill-defined in general. The solution is to re-do the propagation algorithm for any nodes which

```
main() {
  int i,j,x,y;

  foo(&i,&j);                          (1)
  foo(&x,&y);                          (2)
  }

foo(int *a, int *b) {
  *a = *b; }                           (3)
```

Figure 5.8: Example of potential for incorrect flow between function call sites.

could be affected by the goto, after first putting the live assignment list from the goto node to the label node representing the "goto" target. This list then is merged with the live assignment list present at the node from the rest of the propagation algorithm. In practice, this re-processing is of the sub-graph rooted at the least common ancestor of the goto and the goto target in the AST, or, in general, the whole function.

Additional problems come from scope and function boundaries, and how data dependencies are represented and resolved across these boundaries. Especially when considering function boundaries, it is necessary to strictly separate the data-flow graphs. Normal links between the graphs derived from treating function calls as duplicated control structures may result in incorrect flow information, due to false dependencies being found between different call sites of the same function.

For example, Figure 5.8 illustrates a potential false dependence. The value of x after (2) is determined by the assignment at (3). The value of the assignment at (3) is determined by the value of either j or y. Therefore, the value of x is determined by the value of either j or y, producing a flow between x and j which is incorrect. This false dependence is avoided by summarizing the behavior of a function and duplicating this summary information at every call site of the function.

In order to correctly and completely calculate the dataflow information for a single function, all the functions that it calls must be completely analyzed and summarized. If there are no recursive function calls (i.e., cycles in the function call graph), then no additional work is required; the leaves of the call graph are calculated first, and so on up the tree to the root. If there are cycles in the call graph, then the algorithm of Livadas [43] is used to resolve the circular dependencies.

Briefly, Livadas' algorithm iterates through a list of mutually recursive functions propagating more and more reaching definitions until a complete pass through the list produces no new definition-use edges. This algorithm is guaranteed to terminate. In each analysis pass, assignments are only added to live assignment node lists. A pass where no assignments are added terminates the algorithm, and such a pass is guaranteed because the number of assignments is finite. Therefore the algorithm terminates in all cases [43].

5.3.3 Performance bounds

The biggest expense incurred with static analysis is in the live assignment propagation algorithm. The pre-processing step takes time proportional to the construction of the parse tree of the program; construction of the parse tree is common to any compiler or program analyzer, so this portion of the performance bound is not of interest. The interesting portion of the performance bound is the time and space necessary beyond basic parse tree and data-flow graph node creation.

Space bounds

Three factors determine how much space the data-flow graph occupies: the number of nodes, the number of edges, and the size of the live assignment lists. The number of nodes is proportional to the size of the parse tree, with the exception of parameter nodes for global variables. Each global variable used in a function creates an additional node at every call site of the function. Additionally, if one function calls a second function which uses the global variable, then an additional node is added at the call site of each function, so the number of nodes can combinatorially explode for a function that employs many global variables and that has a highly recursive structure.

A data-flow graph is a relatively sparse graph; each node typically has only a con-

	Number of nodes	Number of edges	Total live assignments	Average list size	Maximum list size
lpr	3219	6472	51052	15.86	97
ftp	13397	32731	291016	21.72	249
ypserv	4814	7966	109923	22.83	314
rwhod	1255	3840	25844	20.59	135
at	1460	3226	22573	15.46	129

Figure 5.9: Live assignment list size.

trol dependency edge. In addition, usage nodes typically have one data dependency edge, one declaration edge, and as many data-flow edges as there are alternate assignments to a variable. Only usage nodes, therefore, require more than one edge. Even complex programs require only about twice as many edges as nodes in the graph.

A single live assignment list contains a reference to every assignment currently active; the space for maintaining the lists is the major portion of the size of the data-flow graph. There is no closed-form description of the size of the lists, but for the example programs the number of live assignments at each node is fairly consistent, with between 15 and 23 live assignments per data-flow node on average. Figure 5.9 shows the size of the live assignment lists for the five example programs.

Time bounds

For simple programs with no unstructured jumps and no recursion, relatively tight time bounds can be found. For programs with goto statements or recursion, tight time bounds are not as simple to calculate. The asymptotic bound presented here is conservative; the actual lower bounds are likely tighter. Data gathered from performing static analysis with the Tester's Assistant is used to illustrate the actual run-time behavior.

To calculate the asymptotic bound, a number of variables are used. The values of the variables are calculated with respect to a single function to be analyzed.

N	Number of nodes
A	Number of assignments
E	Number of edges between nodes in the finished graph
L	Number of loops
G	Number of goto statements

Live assignment propagation traverses the static graph in order. For a function without any loops and gotos, the A assignments are propagated to N nodes, and each assignment and every node it propagates to are checked to see if an edge should be created between the node and the assignment. This step takes $O(NA)$. If loops are present, the propagation is repeated for nodes inside the loops. Each node inside a loop gets re-done once, for each loop that the node is in. To simplify the bound, assume that every node is in every loop. An upper bound of this step, therefore, is $O(NLA)$. A `goto` statement propagates the assignment list to the label node that is the destination of the `goto`. The algorithm re-does the traversal for the whole function. This process is repeated until no more assignments can be propagated. The worst case behavior of this algorithm is $O(NLAG)$. Each step of this algorithm must traverse the whole current live assignment list; each step is $O(A)$ in the worst case, so the final time bound is $O(NLA^2G)$

The algorithm is complicated by the interprocedural analysis step if there are recursive function calls. Each function in the recursive call loop is re-analyzed repeatedly until no more assignments are propagated across call sites. So that assignments are not propagated prematurely, all prior assignments to a parameter that could (if analysis of the function is incomplete) provide an assignment are not propagated. As a result, if the parameter turns out to not provide an assignment, a whole list of assignments now needs to be propagated. However, each assignment need only be propagated once to any node, so complexity for assignment propagation can be amortized over several re-analyses of the function. Therefore, if we define P as the number of parameters in all of the recursive call loop, the additional time required for processing the loop is $O(NLGP)$.

Figure 5.10 contains the results of some test runs of the static analyzer on the example Ultrix programs. The tests were run on a Pentium 133 with 32 megabytes of memory. The inner loop count was incremented every time a new node was processed,

	Nodes	Edges	Assigns	Loops	Gotos	Process time – nodes	Average live assigns	Process time – minutes
lpr	3219	6472	280	35	6	5137	15.86	0:07
ftp	13397	32731	1202	114	88	42034	21.72	4:27
ypserv	4814	7966	205	10	0	5601	22.83	0:02
rwhod	1255	3840	66	9	6	3126	20.59	0:13
at	1460	3226	113	14	0	1710	15.46	0:08

Process time indicates how many nodes needed to be processed to complete static analysis.

Figure 5.10: Empirical results from the Tester's Assistant static analyzer.

i.e., every time a live assignment list was propagated to a new node. The reason that the ftp timing results are so different is that the C code for ftp is generated from a yacc grammar. Machine generated code in general is much more difficult to analyze, due to type casting and unstructured jumps.

5.4 Program slicing

An important use of static analysis is program slicing (see Section 3.4). The remainder of this section discusses the production of slices using the data-flow graph. Another use of static analysis is in determining coverage metrics. Chapter 6 describes these metrics.

A basic program slice captures information flow in a program in the same manner as the data-flow graph. Therefore, calculation of a slice once the data-flow graph is constructed is very efficient. A standard backward slice is a transitive closure of edge types representing information flow (control dependency, data dependency, and data-flow); this procedure takes $O(E)$ worst case. The resulting slice represents the entire computation of the value represented by the node upon which the slice was based. A forward slice is calculated similarly, but using these edges in the opposite direction. A forward slice represents the computation influenced by the node upon which the slice was based.

In the Tester's Assistant, each node stores a list of indices to each slice in which it is included. This allows old slices to be re-created by referencing these lists. Then, more complex slices are constructed by combining slices in different ways. This process has been named "dicing" by Livadas [41]. Two basic operators, $\cup$ and $\cap$, union and intersection respectively, are provided to combine slices. These operators examine each node's slice list in turn to determine the complex slice.

Another slice type is a "shadow" slice. A "shadow" of a node in the dataflow graph is the set of all nodes that could precede (for backward shadows, or succeed for forward shadows) the node in an execution. It is important to note that the backward shadow and the forward shadow of a node potentially intersect; a particular block of code can be executed both before and after a program node. For example, if the target node is inside a loop, all nodes within the loop (and possibly others) are in the intersection of the forward and backward shadows.

The various slice types and combination methods are provided as a means of producing more precise slices which are more focused towards a specific goal. For example, suppose a slice was desired that represented all computation between two data-flow nodes that was possibly related to one or both of the nodes. This slice is calculated by the following equation, where *bslice* and *fslice* stand for backward and forward slices and *bshadow* and *fshadow* stand for backward forward shadows:

$$(bslice(node2) \cap fshadow(node1)) \cup bshadow(node2) \cap fslice(node1).$$

Calculating the inverse of this (with *node*1 and *node*2 reversed) and taking the $\cup$ of the two results completely defines how the two nodes interact in the program. This complex slice is a characterization of the TASPEC primitive **before**. The slice defines the ways in which the first node precedes the second node in any execution of the program.

It would be useful to have a full mapping between TASPEC primitives and slice constructors because it would be possible to determine with static analysis the exact program subset which influences a specific property. Without such a mapping, the best that can be done is to take the union of individual slices, which produces a slice

	Program Size	Slicing Criterion	Slice size
Ultrix rdist server	3000+ lines	`chown`	~400 lines
Ultrix fingerd (Internet Worm)	125+ lines	array assignment	~5 lines
Minix login	300+ lines	`setuid`	~20 lines

Figure 5.11: Slicing results.

which is much larger in general than an exact slice would be. Development of the theory of mappings between TASPEC and slice constructors is left for future work.

Recent work by Reps and Rosay [52] suggests some problems with blindly intersecting and unioning slices, especially across call sites. A function that is called in a different location in two different slices appears in each slice. Therefore, the function appears in the intersection of the slices when in fact the individual slices have distinct functionality. To correct for these problems in future work, the $\cup$ and $\cap$ operators need to be refined to function correctly interprocedurally using Reps and Rosay's algorithm.

5.4.1 Slicing results

Figure 5.11 shows the size of slices of some sample programs. The slices are approximately an order of magnitude smaller than the original program. The slicing criteria correspond to the flaws specified in Section 4.6. The size of the slices are approximate because the slice as produced by the Tester's Assistant is formatted differently from the original code.

Chapter 6

Iterative Contexts

Software testing provides meaningful positive results only when some measure is used to indicate how thorough the testing is; a measure is also useful to indicate whether or not the testing is complete. In the context of security and property-based testing, such a measure should also carry with it some relationship to properties; it is desirable to be able, through testing, to declare that a given generic property or flaw does or does not exist in the source code of a given program.

Property-based testing involves primarily structural testing metrics. Structural testing bases coverage analysis on source code, as opposed to functional testing, which bases the analysis on intended functionality. Slicing, the major technique of property-based testing, restricts structural testing to behavior related to given specifications; this restriction is a functional technique. However, the intent of property-based testing is to examine the code for flaws which are independent of the specification of a program; for this examination to succeed, the metrics must require actual paths in the program to be tested. Therefore, successful testing requires covering all paths of interest; computing and covering paths is a structural testing problem.

By using path-based metrics, the Tester's Assistant computes test completeness based upon program slices. It is well-known that complete all-paths coverage is infeasible for practical tests; however, two practical alternative metrics are used in property-based testing. The first metric is complete path coverage of the slice. Experiments with program slicing indicate that the program resulting from the slice is often small enough to overcome this disadvantage (see Section 5.4). However, this metric still requires an infinite number of test cases if loops are in the slice.

The second metric, called iterative context coverage is a new technique developed for the Tester's Assistant. This metric provides some of the benefits of all-paths coverage at less computational expense, although the method also has some limitations. Iterative contexts describes the set of paths to be covered by testing. The thoroughness of a test suite is measured as the percentage of paths from this set which are tested. A complete test suite covers every path from the set. The majority of this chapter is devoted to the construction of the path set using iterative contexts.

One basis of the structural testing metrics presented here is the notion of "context." In general, a context at a point in a program is the set of assignments to variables which are currently live. This chapter refines this notion of context so that a set of required contexts can fully capture a specific computation in the program. The questions that need to be answered are "What variables should be in contexts?", and "How are all the possible contexts for a given requirement calculated?".

In general, coverage metrics *partition* the input space of a program. A given path, for example, is executed only with certain sets of input data. This input data is the partition of the input space defined by that path. Typically, for test data generation, some data is selected from each partition. Path-based coverage schemes, iterative contexts among them, often do not sufficiently partition the input space to provide an effective test [65], especially in cases where the code includes complicated arithmetic computations. The solution is to add additional constraints on the data values of paths. For example, the value 0 as an argument to the `setuid` system call does not necessarily require a different path than an arbitrary natural number as the argument, but the fact that 0 is the user id of the super-user radically changes the effective semantics of the system call. In this case, then, the set of paths which include this system call would be split into two subsets of paths, one of which requires the data value of 0, and the other a positive value. However, further implications of this approach are not discussed in detail in this chapter, as they are outside of the contribution of the work.

In general, the examples in this chapter define coverage metrics in terms of program statements, for the purposes of explanation. In almost any actual implementation of coverage algorithms, the smallest unit for coverage is either a node in the data-flow graph of the program (especially in the case of the Tester's Assistant metrics, which work upon sliced code) or a node in the control flow graph. The algorithms presented should be read in this context.

Section 6.1 describes basic concepts and requirements for coverage metrics. Section 6.2 defines the iterative contexts coverage metric. Section 6.3 describes the algorithm in detail and sketches a proof of the correctness of the algorithm. Section 6.4 compares iterative contexts with other coverage schemes of similar expressive power. Section 6.5 summarizes the chapter.

6.1 Simple coverage metrics

This section defines "coverage metric" and describes several simple metrics. Examining these metrics and their limitations reveals key concepts and requirements for iterative context coverage metrics.

A coverage metric specifies a set of criteria for a test suite to satisfy the metric. One very simple coverage measure is statement coverage. The metric set is the set of all statements of a program; each element (statement) must be executed in order for that element to be satisfied. Thus, complete statement coverage holds if a test suite executes every statement in a program. The measure of thoroughness of the test suite is the percentage of statements that the test suite executes.

The value of a coverage metric lies in its usefulness in forcing flawed code to be executed; statement coverage fails if, for a given flaw to manifest itself, two independent statements must be executed in the same test run. Since complete statement coverage could be achieved by executing these two statements in different executions, statement coverage is insufficient to uncover the flaw.

Hamlet [25] demonstrates this point effectively using a series of simple examples of flawed code. His examples demonstrate the need for a metric more powerful than statement or branch coverage. The remainder of this section presents his examples and the present work's extensions that demonstrate and motivate iterative contexts,

```
shields = 0;
spears = 17950000;
if <NighTime2>
  shields = 5472;
GreeksOut(spears + shields);
```

Figure 6.1: A precursor to the Hamlet example.

the metric used in property-based testing.

The basis of the examples is a system call, `GreeksOut()`. An error occurs when the system call is executed with a given argument (17955472). Moreover, the error can only be detected through execution, so static analysis cannot in itself be effective. Therefore, the goal is, in the set of test suite executions, to cover every possible value for the argument to the system call. In many instances, such as when the value is directly read in from the keyboard, this coverage requirement is not possible; other techniques must be used. However, if user input determines the argument value only through control-flow predicates, every possible value for the argument is still represented in the program; only the selection is dependent upon user input. Here it is sufficient to consider assignments within the program, flagging user input where it appears. In actuality, without user input, a program is likely to be deterministic; such programs are not interesting to analyze.

The three examples are presented in increasing order of complexity. The first fault can be detected through statement coverage. The second is impervious to statement coverage, but is detectable through use contexts. The third is impervious to use contexts, but is detectable by other means; this third example motivates the iterative contexts coverage metric.

In the example in Figure 6.1, the trigger `<NighTime2>` could only trigger in very unusual circumstances; naive testing of the code would not necessarily trip the trigger. Statement coverage would force the statement `shields = 5472;` to be executed for complete coverage, and the fault would be found.

Figure 6.2 is Hamlet's basic example. The erroneous execution occurs only if both predicates `NighTime1` and `NighTime2` are true. A statement coverage metric is

```
spears  = 0;
shields = 0;
if <NighTime1>
  spears = 17950000;
if <NighTime2>
  shields = 5472;
GreeksOut(spears + shields);
```

Figure 6.2: The basic Hamlet example.

```
spears  = 0;
shields = 0;
blade = 0;
shaft = 0;
if <NighTime0>
  blade = 17950000;
if <NighTime1>
  spears = blade + shaft;
if <NighTime2>
  shields = 5472;
GreeksOut(spears + shields);
```

Figure 6.3: An extension of the Hamlet example.

insufficient for this, for the reasons stated above.

However, consider the context of the system call `GreeksOut`; two variables are used in the call. *Use context* testing requires all different combinations of assignments to be executed for these two variables. This coverage metric, then, would require both `spears = 17950000` and `shields = 5472` to be executed in a single test run; this requirement would catch the fault.

Consider the example in Figure 6.3. This example is slightly more complex than the previous example; the use contexts of both `spears = blade + shaft` and `GreeksOut` need to be combined in some way to get a satisfactory metric. Hamlet concedes that there is no solution to the cat-and-mouse game of stronger coverage measures and more obscure code (the locksmith game from Section 3.5 [25]), but has proposed Laski's technique (see Section 6.4.1) as a partial solution.

As we shall see, within certain parameters it is possible to produce bullet-proof coverage metrics, and it is also possible to quantify the error potential in these metrics. These requirements lead to iterative context coverage metrics.

6.2 Iterative context coverage metrics

This section introduces iterative contexts, a new coverage metric. Iterative contexts derive from the identification of sub-computations of a program and require that these sub-computations be covered in a complete test suite. A context is based upon a set of variables at a point in the program, and possible mappings of these variables to the assignments which produced the variables. Contexts are iterative because each of the assignments forms the base for another context, and these secondary contexts are incorporated in an iterative fashion into the base context.

A context is a sequence of assignments in the program, ending at the point of interest. The context requires that the assignments be executed in the order that they appear in the sequence so that the context is covered.

Each context describes a sub-path through the program. A context is satisfied if that sub-path is executed. Testing is complete if and when all contexts are satisfied. However, certain loops are not tested fully with a complete iterative context metric; the situations in which this arises is characterized in general and can be identified specifically when the metric is produced for a specific program.

6.2.1 Loop-free paths; the base case

The base case for deriving iterative path coverage metrics is the algorithm for loop-free paths. A program with no loops has very simple data-flow characteristics; the complete set of paths to test a specific location is finite. Two important characteristics of loop-free programs should be addressed before the more complex case can be handled.

The first is control dependence. Consider the following code:

```
if (c > 0)
   a = b + d;
```

If the variable of interest is a, clearly the context must include both b and d. Additionally, if the new value of a is denoted by a', its value can be represented by the following equation:

$$a' = \begin{cases} b+d & \text{if } c > 0 \\ a & \text{otherwise} \end{cases}$$

This equation shows that the variable c is an element of the computation of a, and therefore assignments to c should be in the context. Inclusion of variables from conditional predicates complicates the calculation of contexts; the additional question that arises, because an assignment may (and almost certainly does, intraprocedurally) be in nested conditional scopes, is: Which of the predicates of these conditionals must appear in the context?

In actuality, variables in control predicates do not need to be explicitly included in the context, because they are implicitly included via the sequencing of assignments in the context. The final assignment in the context is the assignment `a = b + d`; in any execution of this statement, the value of c is greater than zero, so the computation of c which produces this value is implicitly included in a context which ends in the assignment to a.

To illustrate, a short program and the contexts needed to cover the program are shown in Figure 6.4. The first context merely requires the definition of a at statement (1) to be executed, followed eventually with statement (6) and no intervening definition of a. The second context requires the other derivation to be executed in a test run. Combinatorially more contexts are required as the numbers of possible definitions of the variables a, b, and d increase. For example, if another assignment (3′) were possible for d, the additional context 2,3′,5,6 would be required. By requiring (or barring) statement (5) from test executions to satisfy certain contexts, the value of the conditional (4) is automatically tested; in the case of the first context, the conditional fails, and in the case of the second context, the conditional succeeds.

The second characteristic of loop-free paths to consider is input dependence. This characteristic makes testing necessary and also, if present in great numbers, can

```
(1)  a = 1;
(2)  b = 2;
(3)  d = 3;
(4)  if(c > 0)
(5)    a = b + d;
(6)  setuid(a);

Contexts:  (sequences of line numbers)
   1,6
   2,3,5,6
```

Figure 6.4: A short program and coverage contexts.

make generating suitable tests very difficult. However, as above, input dependences for variables in conditional predicates can be safely ignored in the construction of contexts for testing. Only the case of a variable being directly dependent upon input, then, need be considered.

The presence of input data in a context can be abstracted out; in Figure 6.4, if the assignment to `d` is replaced by statement (3′) `d = read();`, the context `2-in-3′-5-6` replaces the second context. The addition of `in` in the context means that, in this location, there is a direct dependence upon input data. Unlike a usual computation in a program, which is deterministic once variables have values, an input can result in any data value. Therefore, merely covering the assignment 3′ is not sufficient to address the goal of validation. More executions are required to test the assignment with different input values.

The suitability of test data to cover the input value, and thus context, then, is directly related to the spread of values desired in the slice criterion, i.e., `setuid()`. Analysis of the domain of possible input values is outside of the scope of the work presented here (see [30] for some approaches to this problem). The Tester's Assistant identifies whether the input-related context is covered (trivially) and leaves further testing of the input domain to the analyst (see page 82). In the case of `setuid()` given in the example, one special value of the argument of interest is the number zero; this value adds another partition to the data space above and beyond the path coverage criteria. These distinguished values can be specified in TASPEC; however, the tracing of these special values backwards through the computation of the program to the input domain is outside of the scope of iterative contexts.

6.2.2 Contexts in the presence of loops

In loop-free code, every backward chain from the use of a variable and a definition of that variable to a use of another variable at that definition and a definition of the second use, etc. is guaranteed to terminate with either a constant assignment or an input as an initial value for a variable. A *recursive computation* occurs when a variable is used in its own re-definition, either directly or in a cycle of definition links. With a loop, recursive computations are possible, so complete contexts are not finitely enumerable in some cases.

Typically, in other testing methodologies, loops are dealt with by requiring them to be executed with different iteration counts: zero, one, and more than one. This control flow requirement catches all definition-use edges in the data-flow graph, but not necessarily the definition-use-definition-use chains of edges. Using contexts to access direct variable dependencies renders pure control loop counts irrelevant, but the loop count technique does transfer over to loops in the data-flow graph.

For some loops, not involving recursive computations, contexts are sufficient to capture dependence information, and special analysis and handling are not needed. For example, many programs have a command loop, in which a user is prompted for a command, the command is entered and performed; the process then repeats. This loop does not in many cases involve recursive computations; but dependencies of state variables accumulate across loop iterations. Figure 6.5 is an abstract example of this phenomenon. The variable `a` is dependent upon `e` only if the loop iterates three times. These dependencies may be accurately captured by contexts that trace dependencies back to their roots. The dependence between `a` and `e` is not captured by control-only loop count methods.

The more complex case is the algorithm for loops involving truly recursive computations, i.e., where there are loops in the data-flow graph . Some provision must

```
while(<expr>) {
   a = b;
   b = c;
   c = d;
   d = e;
   ...
   }
```

Figure 6.5: A loop where dependences build up across iterations.

```
int gcd(int x,int y) {
  while(x != y)
    if(x < y) y -= x;
    else x -= y;
  }
```

Figure 6.6: The gcd program, an example of a numerical algorithm which is hard to test with the Tester's Assistant.

be made for termination; but the simple contexts involving zero or one time through the loop involve the same calculation as that for loop-free contexts. The algorithm to generate iterative contexts terminates calculation after the dependency loop has been traversed twice. Note a twice-iterated loop in the dependency graph might encompass many iterations of the actual control loop, as the recursive assignments may not be activated on every traversal of the actual loop.

Functions implementing numerical algorithms are the most difficult cases for iterative contexts. In the greatest common denominator (gcd) program in Figure 6.6, there are arbitrarily long chains of dependences from x to both x and y, and so on. The lengths of these chains during execution depend on the input values of x and y. The gcd program is an example of algorithmically complex code. In general, structural testing methods like iterative contexts have problems with such code.

Some craftiness on the part of the human tester must be used in concert with the Tester's Assistant's algorithms in the case of truly recursive computations. One can analyze not the computation itself but the control predicate for the loop, and/or the possible initialization values for variables involved in both the predicate and the recursive computation. In general, the tester must realize that the path contexts are insufficient for some criteria (the Tester's Assistant can warn about recursive dependences as well) and then alter the form of the test.

6.3 Calculating iterative contexts

Calculation of iterative contexts for use in coverage metrics is performed by traversing the definition and dependency edges of the dataflow graph. These edges are identical to those used in calculating the slice of a program; however, the calculation of contexts is more complex than the simple graph traversal used in slicing.

To detect possible orderings of execution of different assignments, more detailed live assignment information is necessary. The unit of information needed is

Can this assignment be executed before or after that assignment (or both)?

If the first assignment is still live at the second assignment, then the first assignment can be executed before the second. This information is available as a side effect of static analysis.

The algorithm to construct iterative contexts is described in Figure 6.7. The algorithm expands each assignment in a context by incorporating the context rooted in the assignment. The context rooted at a specific node is determined by the dependences at that node, for example:

assignment	dependences
`a = b + c`	{b,c}
`a = foo(b,c)`	{b,c}
a++	{a}

The contexts are combinations of definitions of the dependence variables. If there are two variables `b` and `c` in the dependence set, and there are two assignments to each variable (b1 and b2, c1 and c2 respectively), then there are four possible contexts: {b1,c1}, {b1,c2}, {b2,c1}, and {b2,c2}. These contexts are subject to possible path constraints; if c2 is always executed before b2, then the context is actually {c2,b2}.

Additionally, if b2 and c2 are on different branches of the same `if` (and the `if` is not in a loop), so that b2 never reaches c2 and vice versa, then {b2,c2} represents an invalid path and so is discarded[1].

Each individual context (e.g. {b1,c1}) is expanded by the contexts of the assignments forming the constituent parts. If b1 represents the code `b = d + e`, then this determines basic contexts for b based upon possible definitions of `d` and `e`. These basic contexts are incorporated into the context {b1,c1} so that the definitions of `d` and `e` occur before b1, but otherwise all possible paths are included. So if there is one definition of `e` but two of `d`, the context {b1,c1} is replaced by the two contexts {d1,e1,b1,c1} and {d2,e1,b1,c1}, assuming that these correspond to possible paths in the program. After this expansion, the definitions d1, d2, and e1 are expanded from their respective contexts in a similar manner.

If the dependency graph has cycles, then the algorithm as stated so far does not terminate. An additional check is made every time a new context is created; if this context has two identical sub-sequences consecutively, then it represents a cycle in the dependency graph that has been traversed two times; such a cycle is called a doubled cycle. A doubled cycle becomes a final context, and is removed from further computation.

For illustration of the algorithm, consider the program in Figure 6.9. Part of the computation of the contexts for the program is enumerated in Figure 6.8. Contexts that are extracted from the calculation as "Final" contexts form the iterative contexts coverage metric for the `setuid` function call in this routine.

Here the correctness of the algorithm in calculating all pertinent contexts for programs with non-cyclic dependency graphs is stated and proven.

Theorem: All computations of data values at node n are covered by a context path in M at the conclusion of the algorithm, excluding computations that have cycles in the dependency graph.

Proof: A different computation which is not in the set of contexts must be de-

[1]Note that the invalid path including b2 and c2 in the code `if (a) b2 = x; if (!a) c2 = x;` is not caught in this step because the Tester's Assistant static analysis does not catch the common sub-expression `a` and treats each branch predicate as a totally separate entity.

1. Initialize $C = \{\{n'\}\}$, the set of working contexts. Initialize $M = \{\{\}\}$, the set of completed contexts. n is the (single) point in the program which the context is based upon. The prime indicates a node whose dependences have not been expanded into the context.
2. Select any initial (primed) node from a context to expand.
3. Run create-contexts() on that node and its dependences to create a new value for C.
4. If there is a doubled cycle (a consecutive sequence of nodes repeated twice) in a context, move that context to M.
5. Eliminate duplicate contexts
6. While C is non-empty, repeat.
7. M contains the required iterative contexts.

```
create-contexts(in-contexts,deps)
  out-contexts = {}
  foreach i (deps)
    tmp-contexts = add-to-contexts(contexts,i,defs(i))
    out-contexts = out-contexts ∪ create-contexts(tmp-contexts,deps - i)
  return out-contexts

add-to-contexts(in-contexts,i,defs)
  out-contexts = {}
  foreach c (in-contexts)
    if some $def$ is already in c to the left of i
      out-contexts = out-contexts ∪ c
    else
      foreach d (defs)
        out-contexts = out-contexts ∪ c ∘ d
  return out-contexts
```

Note: The ∘ operator inserts a node into a context in all legal orderings.

Figure 6.7: Iterative context construction algorithm.

Initial context: $\{12'\}$

Assignments to e at 12 are 5 and 10

$\{5', 12\}, \{10', 12\}$

5 has no dependences

$\{10', 12\}$

Final: $\{5, 12\}$

10 depends on b (2 & 7) and d (9)

$\{2', 9', 10, 12\}, \{7', 9', 10, 12\}, \{9', 7', 10, 12\}$

2 and 7 have no dependences

$\{2, 9', 10, 12\}, \{7, 9', 10, 12\}, \{9', 7, 10, 12\}$

9 depends on a (1 & 11) and c (8)

$\{1', 2, 8', 9, 10, 12\}, \{2, 11', 8', 9, 10, 12\}, \{1', 7, 8', 9, 10, 12\}, \{1', 8', 7, 9, 10, 12\},$

$\{7, 11', 8', 9, 10, 12\}, \{11', 7, 8', 9, 10, 12\}, \{1', 8', 9, 7, 10, 12\}, \{11', 8', 9, 7, 10, 12\}$

1 and 8 have no dependences

$\{2, 8, 11', 9, 10, 12\}, \{7, 11', 8, 9, 10, 12\}, \{11', 7, 8, 9, 10, 12\}, \{11', 8, 9, 7, 10, 12\}$

Final: $\{1, 2, 8, 9, 10, 12\}, \{1, 7, 8, 9, 10, 12\}, \{1, 8, 7, 9, 10, 12\}, \{1, 8, 9, 7, 10, 12\}$

11 depends on 10

$\{2, 8, 10', 11, 9, 10, 12\}, \{7, 10', 11, 8, 9, 10, 12\}, \{10', 11, 7, 8, 9, 10, 12\},$

$\{10', 11, 8, 9, 7, 10, 12\}$

10 depends on b (2 & 7) and d (9)

$\{2, 8, 9', 10, 11, 9, 10, 12\}, \{2, 9', 10, 11, 7, 8, 9, 10, 12\}, \{2, 9', 10, 11, 8, 9, 7, 10, 12\},$

$\{9', 7, 10, 11, 8, 9, 10, 12\}, \{7, 9', 10, 11, 8, 9, 10, 12\}$

... expansion continues of 9 and its dependent nodes:

From $\{7, 9', 10, 11, 8, 9, 10, 12\}$ are produced

Some final contexts:

$\{10, 11, 8, 9, 10, 11, 8, 9, 10, 12\}, \{11, 7, 8, 9, 10, 11, 7, 8, 9, 10, 11, 8, 9, 10, 12\}$

Figure 6.8: Computation of contexts.

```
a = b = c = d = e = 0;      /* Nodes 1-5*/
while (f(e) > 0) {
  if (f(a))
     b = 2;                 /* Node 7 */
  c = 3;                    /* Node 8 */
  d = a + c;                /* Node 9 */
  e = b + d;                /* Node 10 */
  a = g(e);                 /* Node 11 */
  }
setuid(e);                  /* Node 12 */
```

Figure 6.9: Sample program with node labels.

scribable by a sequence of assignments. Thus, the question reduces to: Is there a sequence of assignments not in the set of contexts M (and not involving a dependency loop) which can produce a different result at the target point? Note that the cases where the target point is not executed are not interesting; only sequences of assignments ending in the target point are important. Once this distinction is made, the proof follows quickly; if there is an assignment sequence $L \notin M$, then there is a last assignment l_i (closest to n in the context) such that the suffix of l defined by i differs from every context in M. The assignment l_i, then, must compute some value which is used in some $l_j, j > i$. l_j and its successors in l are a suffix for some sequence in M. However, by the construction of members of M, all nodes which determine values used at a node in the context are included in the context, so l_j in the context implies that l_i is in some context, which means $L \in M$, which contradicts the initial statement that $L \notin M$. QED.

6.3.1 Extension to dual program locations

This algorithm generates contexts for a single program locations. However, many of the specifications to be validated by property-based testing are dual-property. For example, the

$$\mathit{authenticated}(U)\ \texttt{before}\ \mathit{permissions\_granted}(U)$$

specification semantically combines two separate program locations. A coverage metric, to be effective at testing this specification, must accurately capture the context produced by interactions between these two locations.

A simple, although inefficient, solution is to expand the algorithm to multiple program locations by running the algorithm twice, one for each location, and then fully interleaving the resulting contexts, pairwise matching the two sets, to produce many larger contexts. This approach is computationally expensive because of the large number of invalid program paths generated.

Alternatively, fine-grained algorithmic adjustments could enable the two locations to be processed simultaneously. Instead of one initial context $C = \{\{n'\}\}$, use two initial contexts $C = \{\{n', m'\}, \{m', n'\}\}$. In the program in Figure 6.10, neither of the two initial lines of interest, 6 and 10, are executed strictly before the other, so the two initial contexts are $\{10', 6'\}$ and $\{6', 10'\}$.

Finally, the problem of associating two (or more) locations in a coverage context can be addressed by treating assertions as auxiliary program variables during the analysis phase; this potential extension to the Tester's Assistant is discussed in Chapter 9.

6.4 Comparison with other metrics

In this section, the iterative contexts metric is compared to other metrics. The use- and live-contexts of Laski and Korel [36, 39] are similar in form to iterative contexts. Required k-Tuples by Ntafos [49] unrolls def-use chains by chaining together sequences of def-use-def links, and attaches "required elements" as predicates to be satisfied before the chain matches an execution. Finally, iterative contexts are briefly compared with classic path coverage metrics such as all-uses and all-paths.

For comparison, a simplification of the ftpd example from Section 4.6.1 is used. The code for this example is given in Figure 6.10. Various portions of the code have been abstracted away; most significantly, the authentication routine has been distilled into a single call to `match`.

```
(1) logged_in = 0;
(2) while(1)
(3)   switch(cmd) {
(4)     user:  name = read();
(5)            pass = read();
(6)            if(match(name,pass))
(7)              logged_in = 1;
(8)            break;
(9)     get:   if (logged_in)
(10)              setuid(name);
(11)    }
```

Figure 6.10: Abridged source of ftpd.

Testing is based upon the property

$$\textit{authenticated}(U)\ \texttt{before}\ \textit{permissions\_granted}(U).$$

Authenticated is linked to the `match` call, and *permissions_granted* is linked to the `setuid` call. The initial contexts, then, are $\{10', 6'\}$ and $\{6', 10'\}$. The final contexts, after the algorithm is run, are $\{\{4, 5, 6, 10\}, \{4, 5, 6, 4, 10\}, \{4, 10, 4, 5, 6\}\}$. Execution of the path corresponding to the second context results in the erroneous execution.

6.4.1 Use- and live-contexts

The term *context* comes from the work of Laski and the STAD system [39, 36]. Use contexts are similar to iterative paths; at a program assignment, all possible combinations of definitions of variables used on the right hand side form a set of use contexts as a coverage metric. Use contexts, then, are a strict subset of iterative contexts, because the use context forms the basis of a iterative context. Iterative contexts extend use contexts by including all combinations of definitions of the definitions of the variable set, and so on in an iterative process.

Live contexts address the problem that the result of a variable computation also depends upon its context within the program; variables that determine this context can be totally independent of the variables directly used in the computation. Therefore, in a live context, variables which are live (i.e. there is at least one potential reference to the variable in the remainder of the program) are included. Computation of the context from this point on is the same as the use context above.

Live context as a coverage metric ranks above the definition-use based criteria because of this additional context information. However, the number of live contexts in a program is very large; also, the values of the related variables are not important, only that the value comes from a certain assignment. The input stream, then, is difficult for live contexts to handle. The reason that the Tester's Assistant uses iterative contexts over live contexts is that the path requirements produced by iterative contexts are more precise; the only contexts which are produced are those which directly relate to path conditions leading up to the computation that is being tested.

In the case of the ftpd program, the live contexts include the context $\{4, 5, 7\}$. This context is covered by the execution sequence {4,5,7,10}, and also by the sequence {7,4,5,10}, since there is no explicit ordering of statements within the live context. To force the flaw to be discovered, the second sequence must be required, but live contexts allow the context to be covered by the first sequence. Iterative contexts do require the second sequence, and so detect the flaw.

6.4.2 Required k-tuples

A k-tuple is a definition-use chain of length k in a program. A k-tuple is linked by explicit information flow; each definition is used in the successive definition. A required k-tuple consists of such a chain along with an assertion about the values of variable at that point in the program. More specifically, the k-tuple specifies, or partially specifies, a path through the program; the assertion then puts an additional requirement at the end of the path. The path must be executed with the requirement holding in order for the test execution to cover the required k-tuple. In practice, the requirement assertions are conditional predicates (or their negation) from the program location. Therefore, the assertions can serve as additional information in the partitioning of the input domain.

While the general model of k-tuples seems general enough to capture many generic requirements and TASPEC specifications, in practice many extensions are necessary. Path expressions (sequence of program nodes) are limited; accurate capture of a data context requires several definition-use chains; capture of several different chains with k-tuples is problematic. This difficulty is due to the restriction that k-tuples explicitly represent information flow; in an iterative context, definitions can be interleaved independently of explicit dataflow. In the ftpd example, the context $\{4, 5, 6, 10\}$ is not a k-tuple because the node at 6 does not define a value used at 10; no explicit information flows between 6 and 10.

In this regard, relaxing the k-tuple restriction allows monitoring of higher-level semantic behavior of the program. In the case of the ftpd example, the two code locations could not be tested simultaneously, because the semantic information represented by *permissions_granted* and *authenticated* is not associated with explicit program syntax.

The k-tuple handles loops in a different way than iterative contexts do. Definition-use chains are used up to depth k, regardless of the presence of loops. Depending on how complex the intra-loop dependences are, depth k may or may not define more paths than do the limiting recursive simplification used in iterative contexts.

6.4.3 Path coverage

The motivation for path coverage metrics is the same as the motivation for the Tester's Assistant metrics. Simple metrics are obviously insufficient, but all-paths is obviously impractical. The basic difficulty that other path- or context-based metrics have remains the correlation of uses and definitions of a variety of variables. Of the simpler path coverage metrics, all-uses is the strongest. All-uses requires execution of every definition-use pair. As we see from the sequence of examples, coverage of def-use pairs in itself is not sufficient, as all possible combinations of definitions of two variables at a point in the program are necessary.

All-def-use-paths requires every simple path from every definition to every use to be executed; this requirement is only trivially less expensive than all-paths. However, all-def-use-paths does not detect the ftpd fault; the path $\{4, 5, 6, 4, 10\}$ is not a simple path, and so does not appear in the all-def-use-paths criteria. Once again, the iterative coverage metric more exactly fills the requirement to detect specific program faults.

6.5 Summary

Structural testing criteria are necessary for testing to validate properties of code. Previous structural testing criteria have serious problems with programs with loops and do not address some requirements for validating straight-line code.

Iterative contexts solve the problem for straight-line code, in fact for any code without loops in the dependency graph. Therefore, the iterative contexts coverage metric is strictly stronger than other structural coverage metrics. Combined with analysis of the input space, iterative contexts are sufficient to provide a validation of a property.

Code with loops in the dependence graph remains a problem for structural testing. Better techniques need to be developed to cope with these sorts of functions. For now, the best solution is to identify functions with recursive loops and rely on a human analyst to validate this code.

Chapter 7

Execution Monitoring

The two goals for the Tester's Assistant execution monitor are establishing the correctness of a single test execution and establishing the completeness of a set of test executions. Correctness and completeness are defined with respect to property specifications. The C source code of the program is instrumented to measure correctness and completeness. Then, the altered source code is compiled, executed, and tested.

An execution of a program is correct if it does not violate the property specification. A set of executions is complete if they satisfy the iterative context coverage metric.

Execution monitoring in the Tester's Assistant has the following characteristics:

1. TASPEC specifications guide automatic generation of audit trails of interesting behavior.

2. An event engine generates high-level events from audit trails. The events are checked for correctness with respect to the TASPEC specifications.

3. Execution traces are compared with iterative contexts to determine coverage results.

4. Audit and trace information needed for execution monitoring are generated by instrumenting the monitored program.

Section 7.1 describes correctness monitoring, the requirements to generate the audit events, and the system to evaluate the audit events. Section 7.2 describes completeness monitoring and the use of trace information to calculate completeness. Section 7.3 describes the source code to source code transformation to generate the required audit and trace information for the two types of monitoring.

7.1 Correctness monitoring

Correctness monitoring has three aspects: calculation of the information to place into the audit trail, generation of the actual audit trail, and analysis of the audit trail. The composition of the audit trail elements is defined with TASPEC specifications. Instrumentation of the target C program produces a modified program that generates the audit trail during execution. Section 7.3 covers details of the instrumentation. The audit trail is examined to see if any error conditions exist; examination is done according to the semantics of TASPEC.

Property-based testing of the Minix login program [62] and the authentication property illustrates correctness monitoring. Given a specification such as

$$authenticated(U)\ \textbf{before}\ permissions\_granted(U),$$

the question is "How is satisfaction of this invariant determined during program execution?" TASPEC provides location parameters with which primitive events are associated with specific program locations.

Consider the specification for the `authentication` property, shown in Figure 7.1. The specification defines three primitive events and one higher-level event. These primitive events are passed as they occur in the execution to the execution monitor. The trace of primitive events is equivalent to application- or specification-level auditing.

The execution monitor records the running state of the program with respect to the specified events and predicates. When a combination of predicates (with appropriate argument values) occur in the state (as specified by the invariant (4)) higher-level predicates such as *authenticated* are asserted.

(1)**location func** *crypt*(*password*, *salt*) **result** *encryptpassword*{
  assert *password_entered*(*encryptpassword*); }

(2)**location func** *getpwnam*(*name*) **result** *pwent*{
  assert *user_password*(*name*, *pwent* → *pw_passwd*, *pwent* → *pw_uid*); }

(3)**location func** *strcmp*($s1, s2$) **result** 0{
  assert *equal*($s1, s2$); }

(4)*password_entered*(*pwd*1) ∧ *user_password*(*name*, *pwd*2, *uid*) ∧ *equal*(*pwd*1, *pwd*2){
  assert *authenticated*(*uid*); }

Figure 7.1: Property specification of the *authenticated* predicate.

(5)**location func** *setuid*(*uid*) **result** 1{assert *permissions_granted*(*uid*); }

Figure 7.2: Specification of *permissions_granted.*

The *permissions_granted*(U) predicate is less complicated, because only one primitive event, raised by the `setuid()` call, is used (See Figure 7.2).

Finally, because of the top-level invariant

(6)*authenticated*(U) **before** *permissions_granted*(U),

the event engine checks the state every time the *permissions_granted* predicate is raised. If a corresponding *authenticated* predicate is not found in the state, the program execution is flagged as being in error.

Using this specification, an execution of the login program is monitored for correctness. Figure 7.3 shows relevant portions of the source code. The first location matched is `getpwnam(name)`, which has location specifier number 2, one parameter, and two significant elements of the return value structure. Assume in the test execution that the value of `name` is `gfink`, the value of `pwent->pw_passwd` (the encrypted password) is `o1tAXz9qdbutk` and the value of `pwent->pw_uid` is 287. Then the following audit event is created:

```
2 "gfink" "o1tAXz9qdbutk" 287.
```

The next location to execute is the `crypt` call. Assuming that the correct password is entered (the encrypted password in the password file), the audit event is:

```
1 "o1tAXz9qdbutk".
```

Next, the `strcmp` function call is executed, and if the return value is 0, as is the case in this example, the audit event is:

```
3 "o1tAXz9qdbutk" "o1tAXz9qdbutk".
```

According to the login program, this is a correct authentication, so permissions are granted to the user, by the system call `setuid`. Thus, the final audit event generated is:

```
5 287.
```

A TASPEC engine calculates the correctness of the execution based upon the audit trail. The first audit event results in *user_password*(*gfink*, `o1tAXz9qdbutk`, 287) being added to the abstract state database. After the predicate is added to the state, the state is checked to see if any invariants are violated or any pre-conditions are matched. Due to cross-references from facts to these formulas, only the pre-condition (4) is checked, as that is the only formula in which *user_password* appears. The pre-condition is not satisfied, so no further action is taken.

The second audit event results in the predicate *password_entered*(`o1tAXz9qdbutk`) being entered. Once again, the pre-condition (4) is checked, but it still is not satisfied, so the TASPEC engine proceeds to the next audit event.

When the third audit event is processed, the predicate

$$equal(\texttt{o1tAXz9qdbutk}, \texttt{o1tAXz9qdbutk})$$

is added to the database. This time, pre-condition (4) does match, with the variable bindings:

variable	pwd1	name	pwd2	uid
binding	o1tAXz9qdbutk	gfink	o1tAXz9qdbutk	287

```
main(argc,argv)
int argc;
char **argv;
{
  int bad, n;
  char name[30];
  char password[30];
  struct passwd *pwd;

  for (;;) {
        bad = 0;
        if (argc > 1) {
                strcpy(name, argv[1]);
                argc = 1;
        } else {
                do {
                        write(1, "login: ", 7);
                        n = read(0, name, 30);
                } while (n < 2);
                name[n - 1] = 0;
        }
        if ((pwd = getpwnam(name)) == NULL) bad++;
        if (bad || strlen(pwd->pw_passwd) != 0) {
                write(1, "Password: ", 10);
                n = read(0, password, 30);
                password[n - 1] = 0;
                write(1, "\n", 1);

                if (bad && crypt(password, "aaaa") ||
                    strcmp(pwd->pw_passwd, crypt(password,
                                                pwd->pw_passwd))) {
                        write(1, "Login incorrect\n", 16);
                        continue;
                }
        }
        setgid(pwd->pw_gid);
        setuid(pwd->pw_uid);
}}
```

Figure 7.3: Partial text of the MINIX login program.

Therefore, the predicate *authenticated*(287) is added to the database. The predicate *authenticated* appears in the invariant (6), so that invariant is checked, and it is not violated.

Finally, the last audit event is processed. The predicate *permissions_granted*(287) is added to the database, and invariant (6) is checked. It is satisfied, and so no error condition is raised. This ends the audit stream, so the program has finished a successful execution without error.

If during analysis of the audit trail any invariant was violated, then the human tester would be informed of the test failure, and then the tester could take whatever steps were appropriate to report and/or fix the fault.

The TASPEC engine for analyzing audit trails is not currently implemented. The two main options for this implementation are: a module written in C that is linked with the program that processes events and flags bad executions, or a separate Prolog processor that processes the primitive events off-line. The Prolog processor is preferable because of the correlation between TASPEC semantics and Prolog pattern-matching and backtracking semantics. By explicitly mapping TASPEC predicates to Prolog rules, the TASPEC engine can be implemented efficiently and effectively. It is not anticipated that implementing the engine will not raise any significant technical issues.

7.2 Completeness monitoring

Completeness monitoring measures the degree to which a program execution satisfies the iterative contexts coverage metric. A context is a sequence of assignments that defines a sub-path through the program. An execution satisfies the context if it executes these assignments in order without any interfering assignments that intervene.

A trace of the program is analyzed to calculate if the execution satisfies any contexts. The trace is not a full instruction sequence for the program; only nodes that are in contexts are included in the trace. Thus, completeness is monitored with respect to particular contexts, which in turn have been determined by dataflow

```
(1) logged_in = 0;
(2) while(1)
(3)   switch(cmd) {
(4)     user:  name = read();
(5)            pass = read();
(6)            if(match(name,pass))
(7)              logged_in = 1;
(8)            break;
(9)     get:   if (logged_in)
(10)              setuid(name);
(11)    }
```

Figure 7.4: Fragment of source of ftpd.

analysis and slicing.

Completeness monitoring has three aspects: deriving the coverage metric, producing the reduced traces, and calculating what traces cover which contexts. Chapter 6 describes the derivation of the contexts. Section 7.3 describes the instrumentation of C programs to produce traces. The remainder of this section describes matching traces to contexts through use of the example fragment of ftpd in Figure 7.4.

For the source fragment, the data-flow graph has 32 nodes. Figure 7.5 displays the data-flow graph. Only four of these nodes are included in contexts in the coverage metric. As previously calculated, the metrics by line number are

$$\{\{4,5,6,10\},\{4,5,6,4,10\},\{4,10,4,5,6\}\}.$$

Line numbers are translated to nodes as follows:

Line number	4	5	6	10
Node number	7	10	15	22

Therefore the contexts as labeled by node numbers become

$$\{\{7,10,15,22\},\{7,10,15,7,22\},\{7,22,7,10,15\}\}.$$

Only nodes 7, 10, 15, and 22 are important, because they are the only nodes in the coverage metric. Therefore, they are only nodes that appear in traces of the ftpd

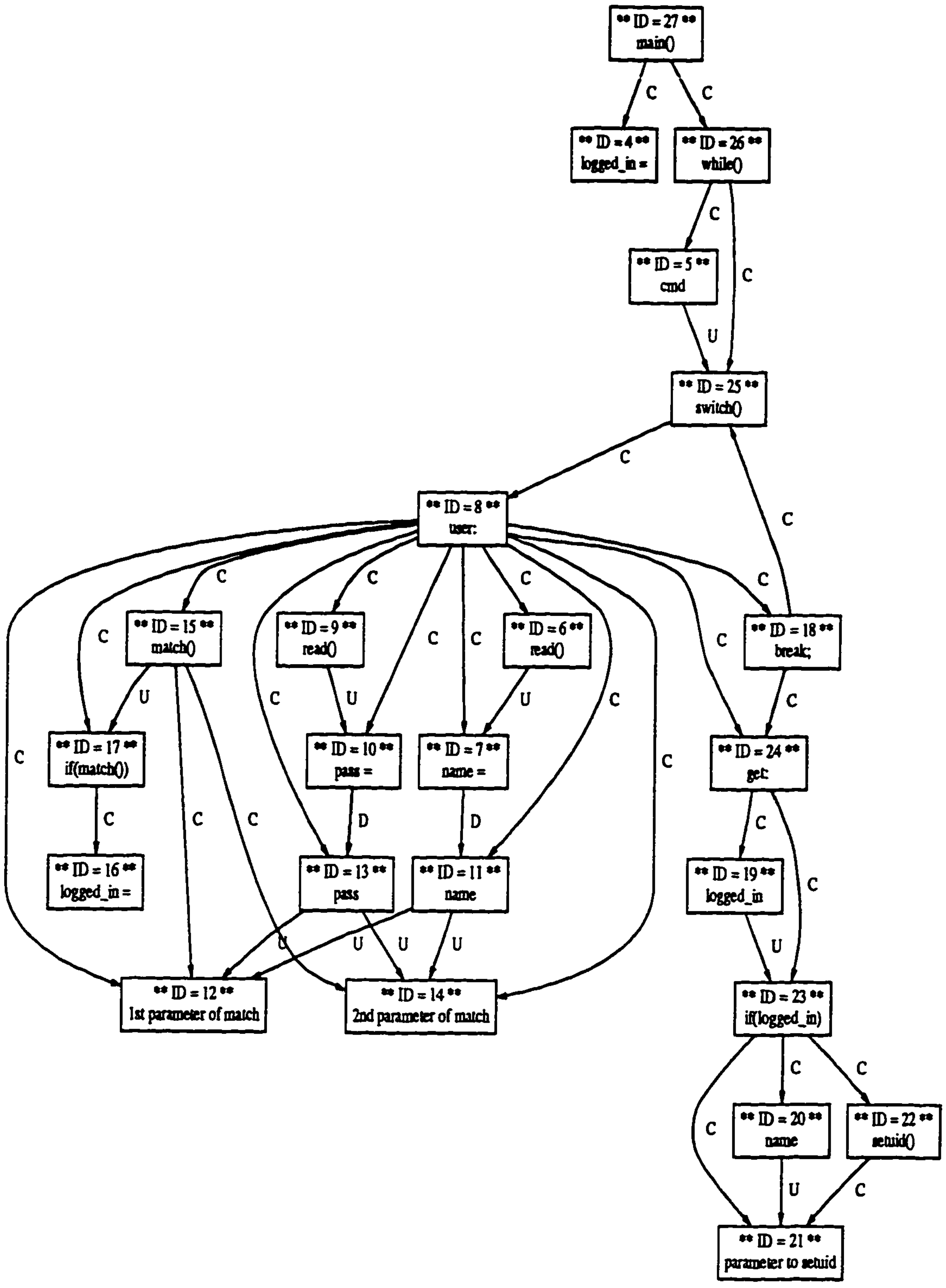


Figure 7.5: Data-flow graph for ftpd fragment.

program. The trace represents the order in which these nodes of the program are executed.

There is not a one-to-one mapping between traces and contexts. A trace may satisfy no contexts, or it may satisfy several. For example, the trace $7, 10, 15, 22, 7, 22$ satisfies each of the first two contexts. But the trace $7, 10$ (corresponding to a failed login) does not satisfy any of the contexts.

Trace analysis uses data-flow and live assignment information, the same information used used in the construction of contexts. Therefore, the trace analyzer is another module in the static analysis engine.

Because of possible intervening assignments, simple pattern matching is not sufficient to match traces to contexts. Each assignment in the context has nodes to its left which represent the dependencies for that assignment. For a trace to match the context, there can be no node in the trace that represents a definition of a dependence of an assignment appearing between the dependence node and the assignment node in the trace. The nodes which can interfere with the dependences for an assignment can be enumerated; they are all the possible assignments to all the possible dependences of a node.

A trace matches a context if the context is a (not necessarily consecutive) sub-sequence of the trace. There is an additional restriction that there be no duplicates of nodes within the area of the sub-sequence that matches the trace, and also that there are not any re-definitions of any variables in the context within this area. For example, suppose the trace $\{7, 10, 7, 22, 15\}$ satisfies the second context but not the first. In the first context ($\{7, 10, 15, 22\}$), the value of `name` generated by the first node in the context is used at both 15 and 22. In the trace, after 15 is executed, `name` is re-defined by another execution of 7. Therefore, the trace does not match the first context.

An algorithm for matching traces to contexts has not been developed; while the case of evaluating whether a single traces matches a single context has been described here, pairwise comparison of traces with contexts is inefficient. Development of algorithms that compare a trace with multiple contexts in parallel is left to future work.

```
{
  if (setuid(u)) {
    ...
    }
}

#define node37 37
{ int tv001,tv002;
if((tv001 = setuid(tv002 = u),
    TRACEEMIT(node37),
    tv001 == 1 ? EVENTEMIT(6,tv002) : 0,
    tv001)) {
  ...
  }
}
```

Figure 7.6: Example of code instrumentation, before and after.

7.3 Instrumentation

Instrumentation of the C code [31] produces the data for execution monitoring (primitive events and traces). Expressions containing pertinent node locations are altered to emit the monitoring information without changing their normal functionality. The Tester's Assistant emits C code as output, the instrumentation is expressed with the conditional expression construct and auxiliary variables. Figure 7.6 shows an example code fragment and its instrumentation.

There are two different monitoring streams: the stream for traces and the stream for events. Every time a data-flow graph node in a context is executed, a reference to that node is placed in the trace stream. When a TASPEC location matches, an event is placed in the event stream.

The instrumentation preserves the semantics of the original code. Sub-expression results such as return values and parameters are saved into temporary variables, which are used in the code for emitting events and traces. Therefore, any expressions in the original program are only executed once, so that side effects in the original expressions or function calls are not executed twice.

In the example in Figure 7.6. the result of the `setuid` system call is saved, both for use later in the instrumentation and for its eventual usage as the predicate for the `if` conditional in the original program. The parameter to `setuid` is similarly saved into a temporary variable. Then, the node number of the `setuid` node is emitted as part of the trace being produced. Next, the return value which has been saved off then is compared to the result portion of the location specification, and if the comparison is successful, the primitive event *permissions_granted* is raised with the correct parameter. Finally, the return value of the original call is returned, and the original source code proceeds as if the auxiliary processing had never been performed.

This method of instrumentation does not affect the normal semantics of the program unless the program is ill-behaved, although it will affect the timing properties of the program. If the timing properties of the program are important, then the probe effect introduced by this instrumentation will interfere. Results elsewhere [4, 66] indicate that it is possible to remove the probe effect in most situations by minimizing the number of probes and intelligently placing the probes in places such as `NOP` instructions following branch instructions.

Chapter 8

The Tester's Assistant

The current implementation of the Tester's Assistant includes a static analyzer for C, a slicer, a TASPEC parser, and a data-flow instrumenter. The data-flow instrumenter was described in Chapter 7. The correctness monitor and the coverage analyzer are not currently implemented, so this chapter focuses on static analysis and slicing.

Section 8.1 describes how a user operates the Tester's Assistant. Section 8.2 provides some documentation of the implementation. Finally, Section 8.3 discusses future directions of the Tester's Assistant implementation, both short and long term.

8.1 Operating the Tester's Assistant

This section describes how to prepare a program and how to perform the analysis. Additionally, this section describes the various forms output can take.

8.1.1 Pre-processing the program

The Tester's Assistant does not maintain any persistent data objects across invocations of the Tester's Assistant, so all related code for a program must be analyzed at the same time. This restriction results in a relatively hard limit ($\sim$ 3000 lines) on the size of program that is able to be analyzed; this restriction will be relaxed later. For

-s name	Function call on which to slice
-v name	Variable on which to slice
-pp	Print out a copy of the original program
-ps	Print out the sliced code
-pi	Print out an instrumented slice
-batch	Run in batch mode
-lf function-spec	Specify the data-flow behavior of a library function

Figure 8.1: Command line arguments for the Tester's Assistant.

now, however, the first step towards processing a program with the Tester's Assistant is to merge all the separate source files into a single file containing all the source code.

Once the source code is all in one file, the file is run through a C pre-processor, so when the Tester's Assistant processes the file, it has labels for external global variables and procedures, so that they can be handled in a consistent manner in the various instances throughout the file. An additional reason is that all the macro expansions will be performed.

Unfortunately, once the pre-processing operation has been performed on a file, the format of the file is often quite different from the original format. The different format can cause problems for the human tester because during testing and debugging it is often necessary to make a correspondence between the code being examined and the original source code. The mapping can be obfuscated by the changed format of the pre-processed file.

After the merging of the multiple files and pre-processing of the result has occurred, the program is now ready to be analyzed by the Tester's Assistant.

8.1.2 Running the static analyzer

The Tester's Assistant is run in either batch mode or interactive mode. Interactive mode should be used initially, trying out various parameters to determine the exact slice desired. Then, the slice can be obtained by redirecting output of the batch mode.

Figure 8.1 contains the command-line arguments for the Tester's Assistant. The first two arguments specify the slicing criterion. Note that the additional slicing primitives (such as ∩ and ∪) are not supported in this interface; use of these requires the interactive interface. The next three arguments specify the output type. An "instrumented" slice monitors trace and event activity during program execution.

The final command line argument, `-lf`, specifies the data-flow behavior of an undefined function. If a function definition does not appear in the file being processed, its parameters are assumed to be both input and output parameters. If a file contains many undefined and unspecified function calls, any slice made of the file is likely to be much larger than necessary. For example, if `foo(c)` is an undefined function call, then the static analyzer makes the worst case assumption that, if `c` is an array, it is assigned to in some invocations of `foo`, but not others. Any weaker assumption could result in a data-flow graph that is missing a definition-use edge, which is unacceptable. Therefore, all assignments before the call are still valid, and another potential assignment to `c` is made. Additionally, if the undefined function call is actually `foo(c,d)` then the side effect assignment to `d` is data-dependent upon the prior values of both `c` and `d`. This, potentially, exponentially increases the size of the slice by adding false dependences.

A function specification, if present, is used instead of this default behavior for an undefined function. A function specification has the following form:

```
<function name>/[IOB]*.
```

Each letter after the "/" specifies the behavior of one of the formal parameters to the function. "I" indicates an input-only parameter, "O" an output-only parameter, and "B" a parameter used both for input and output. Any output parameter (including the return value of the function, which has no explicit specification attached to it) is dependent upon every input parameter (the course granularity of functions will be removed in future versions of the Tester's Assistant). Therefore, the specification

```
foo/IBBO
```

means that the function "foo" has four parameters. The first parameter is input only, the next two are both input and output, and the fourth is output only. The value of the last three parameters, as well as the return value of foo, are all dependent upon the initial values of the first three parameters.

The Tester's Assistant has no scheme at present to keep a catalog of function specifications so that system libraries can be specified in a standard re-usable way. Since the format of the function specifications will change in newer versions of the Tester's Assistant (to add more detailed dependence information, including references to global variables when necessary) to be consistent with the format of the new function summaries, current function specifications will have to be re-created anyway. Ideally, because the new version of the Tester's Assistant will work with multiple input files, it may be possible, through analysis of operating system source code, to automatically generate function specifications, at which point the separate concept of function specifications will become redundant and disappear.

8.1.3 Getting output from the static analyzer

Once the program is analyzed, and the interactive mode is chosen, a user of the Tester's Assistant is prompted for commands to slice the program and combine slices in various ways. Figure 8.2 gives the command set for the interactive mode. Each option type will be briefly described.

Section 5.4 describes the slicing and shadowing commands. When a slice (or shadow) is calculated, it is given a slice index. This index is stored along with a text description of the calculation of that slice. Information is also kept in the data-flow graph recording which nodes are in which numbered slices. Via the "replay" command, old slices can be viewed again.

The "inst" command generates data-flow and event instrumentation. The whole program is printed out, but the portion of the program representing the slice has data-flow instrumentation added. Event instrumentation is done with respect to property specifications which are given to the Tester's Assistant along with the source code.

Appendix B presents the script of an interactive session using these commands.

quit	exit the slicer
back	subsequent slices will be backwards slices
forward	subsequent slices will be forward slices
shadow	turns on shadow mode
noshadow	turns off shadow mode
slice	subsequent operations will be slices (default)
inst	output instrumented rather than sliced code
prog	print out original code
status	display which node numbers are in the slice
summary	print out a description of previous slice commands
replay n	replay a previous slice command (see summary), previous command chosen by n
var id	slice program with respect to the variable id
func name	slice program with respect to the function name
union n1 n2	perform the union of 2 previous slices n1 and n2
intersect n1 n2	perform the intersection of 2 previous slices n1 and n2

Figure 8.2: Interactive Tester's Assistant commands.

8.1.4 Property specifications in the Tester's Assistant

TASPEC specifications given in the grammar described in Chapter 4 can be added to the beginning of code to be analyzed by the Tester's Assistant. If specifications are to be added, the five character sequence "=====" must appear both before and after the text of the specifications. After the second five character sequence is the whole source code of the program. This additional step of manually concatenating specification and code (with separators) is another artifact of the current implementation; as the Tester's Assistant becomes a more polished tool, the specification selection mechanism will become more automated.

8.2 Internals of the Tester's Assistant

The Tester's Assistant is implemented using the ELI [22] language translator toolkit. ELI provides scanning, parsing, and attribute grammar packages, as well as additional utility packages, all unified with a standard interface. In addition to all these basic parts of ELI, the Tester's Assistant uses a syntactic specification of ANSI C in ELI, provided by the creators of ELI.

The Tester's Assistant uses many ELI languages with which parts of the translation process can be specified. *Gla* is the scanner language, and *con* is the parsing language used by the Tester's Assistant, for example. The ELI engine takes these specifications and converts them into a version of the specifications written in C, which is then compiled to produce the final translator. The most important and powerful language is *lido*, in which attribute grammars can be specified. *liga* translates *lido* specifications into executable code, resolving attribute dependencies and scheduling attribute computations with a greedy algorithm.

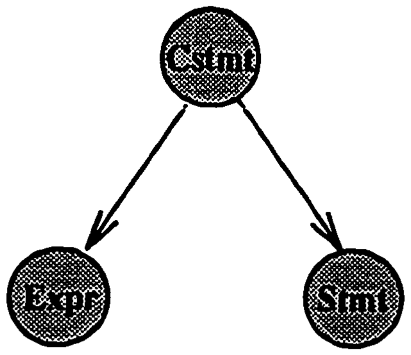
The remainder of this section consists of three parts. The first subsection explains in general the power and limitations of *lido*'s specification of attribute grammars. The second subsection describes the large scale architecture of the Tester's Assistant as it currently exists. The last subsection describes the implementation of data-flow analysis as it was influenced by ELI, and some ways the limitations can be overcome in future versions.

8.2.1 Lido overview

Lido's attribute grammar operates on an abstract syntax tree (AST). The abstract syntax rule for the "while" statement in the Tester's Assistant's AST and its corresponding tree structure is shown in Figure 8.3. "CStmt" is a control statement node, and "Expr" and "Stmt" refer to generic expression and statement nodes respectively. Appendix C describes the complete Tester's Assistant abstract syntax tree for C programs. Figure 5.5 is an sample AST for a small code fragment.

Each node in the AST has many attributes associated with it. Figure 8.4 lists AST attributes and their meanings. There can only be one assignment to any attribute, and this assignment is only executed once. The simple case is for attributes to be assigned within a RULE computation:

```
Expr.ddep = ListPL(CStmt.DFNode);
```



```
RULE rule_159:
CStmt ::=  'while' '(' Expr ')' Stmt
END;
```

Figure 8.3: Sample AST rule.

The "ddep" attribute of Expr is derived by application of the ListPL[1] function to the "DFNode" attribute of Cstmt. Attributes can either be synthesized (bottom up) or inherited (top down). Attributes can also be defined in symbol contributions where attribute values depend wholly on other attribute values at the same node, for example:

```
SYMBOL CStmt COMPUTE
  THIS.DF_node = mk_ctrl_node(THIS.DFNode,THIS.cdep);
END;
```

In addition to the localized access, there are three mechanisms for accessing attribute values from AST nodes which are neither direct ancestor or direct descendants. `INCLUDING` selects an attribute from the nearest ancestor of a specified type. `CONSTITUENTS` collects attribute values from all descendants of a specified type. `CHAIN` allows a value to be accumulated as the tree is traversed in a given order, propagating the value from node to node. In a `CHAIN` computation, all children need to be processed before the next sibling of a node is processed, but the evaluation order for the children can be changed as needed.

These attribution mechanisms only allow attribute equations based upon the

[1]The ListPL function creates a list out of a single element. Other functions which act on this data structure include ConsPL and AppPL (for append).

Var	Description of the variable described or references by the AST node
DFNode	Data-flow node attached to that node of the AST
ddep	List of the data dependences of this node of the AST
cdep	List of the control dependences of this node of the AST
stmtlist	List of AST nodes in order of execution
DF_node	Flag indicating that static edges determined by cdep and ddep have been added to the data-flow graph at that AST node
DF_ready	Flag indicating that the data-flow graph is ready for the live assignment propagation phase
DFgraph	Flag indicating that the data-flow graph construction is complete and that slicing, instrumentation and other operations can proceed

Figure 8.4: AST attributes.

structure of the AST. The Tester's Assistant overlays the structure of the data-flow graph on top of the AST; to derive the data-flow graph it is necessary to track attribute dependences between nodes arbitrarily disconnected in the AST. Because it is not possible to express data-flow dependencies with attribute computations in *lido*, this computation must be done in auxiliary C functions.

Auxiliary C computation is also needed because data-flow graph construction is an iterative process; live variable propagation information is set and re-set multiple times before the final value is found. This computation cannot be expressed given the assign-once limitation of *lido* attributes. Newer versions of ELI (to which the Tester's Assistant has not yet been converted) have a new attribution construct ITERATE, which allows the attribute computation in a subtree to be repeated until a final value is stabilized for an attribute in the root node of the subtree. Future versions of the Tester's Assistant will take advantage of this feature to simplify attribute computation by moving much of the functionality of the C auxiliary functions to lido.

8.2.2 Tester's Assistant architecture

The major portion of the current implementation of the Tester's Assistant is in the static analyzer, including C parsing, symbol construction, and data-flow graph construction. The result of the static analysis is a data-flow graph; currently there is no facility to store a data-flow graph and all of its attached information out to a file, so any of the other modules must be linked with the static analyzer to operate.

Specifically, for slice and TASPEC instrumentation, both the data-flow graph and the AST of the source program are utilized. Nodes in the data-flow graph are targeted for instrumentation, and therefore the AST nodes corresponding to the data-flow nodes are also selected. Complicated output functions using the ptg and lido languages in ELI produce syntactic C code from the AST based upon the targeted nodes that are selected.

The Tester's Assistant's major modules are:

- C parsing: Produces the AST from the near-ANSI C grammar. The official ANSI grammar was altered in two ways. First, C is not LALR(1) because of ambiguity between typedef identifiers and normal variables. Backtracking the token mapping is needed to remove this ambiguity. Second, in the ANSI grammar, the same non-terminal is used for expressions like a++ and `a()`. These two phrases have radically different data-flow interpretations, so the grammar was altered so that different non-terminals refer to the two phrases.

- Output generation: Generates functions to produce valid C code from the AST. This module is mostly automatically generated from the input grammar by the ELI-based "Idem" tool. Small changes to add white space are necessary in the output functions to produce nicely formatted code.

- Symbol construction: Tags each variable (simple or complex) with the memory element it accesses. For example, `a[i]` defines a different (though intersecting) memory reference than does a. The symbol representation is augmented by comparison operators that determine if the memory represented by two symbols potentially overlaps, or if one of the memory images is a subset of the other.

- Data-flow graph construction: Sets up the static dependences and live assignment propagation (see Chapter 5). This module is described in some detail in the above chapter, but some special implementation considerations are described in the remainder of this section.

- Slice creation and instrumentation: Includes special formatting for printing subsets of the AST. For backward slices the rules are simple, because slices are always subtrees of the AST which include the root of the AST. For forward slices and other more complex slices, a node is not always in the slice if some of its children are in the slice. Because the output is constructed bottom up, the fact that the parent node (of a node in the slice) is outside the slice complicates slice construction.

- Specification processing: Attaches primitive events to code locations. Only function and variable code locations are currently supported.

8.2.3 Implementing data-flow analysis in ELI

The Tester's Assistant implements data-flow analysis by attaching powerful attribute computations to AST nodes. Fine-grained analysis where the unit of the data-flow graph is an AST node means that the intricacies of C can be clearly represented. The semantics of C allow arbitrary side effects and global variables, and the execution order of sub-expressions is not well-defined. To represent these features of C, extremely fine-grained data-flow information is necessary.

ELI provides very powerful tools for analyzing programs, and solves a lot of hard issues, especially with the C grammar provided (including typedef disambiguation). Using the ELI C grammar enables the generation of an AST which is optimized for data-flow calculation in the Tester's Assistant. The ability to specify attribute grammars with the lido inheritance mechanism is also very useful because constructing the data-flow graph involves identical computations at many different nodes in the graph. Attribute computations are specified on a rule-by-rule basis, but the overall scheduling of attribute evaluation is done by liga, the lido scheduling engine. So much of the control structure of the Tester's Assistant is automatically generated by ELI.

Because of the assign-once limitation for attribute values, however, in the current version of the Tester's Assistant, much of the real computation (live assignment propagation) must be performed in C code. This computation very naturally lends itself to tree-based computation, because it is highly dependent upon execution ordering.

In live assignment propagation, a cumulative list of live assignments is transferred from data-flow node to data-flow node in order of evaluation. This order is directly derived from the structure of the AST; the information must be transferred into C data structures; this extra structure makes the data-flow graph representation and calculation more complex than it need be.

With the ITERATE construct, live assignment propagation can be moved to lido. The data-flow graph representation can be simplified by removal of order information. In the current graph, control nodes are differentiated by type (while, for, etc. are all different types); this will disappear in newer versions of the Tester's Assistant. As a result, the internal form of the data-flow graph will become more language independent, resulting in opportunities to apply the Tester's Assistant to different source languages.

8.3 Future directions for the Tester's Assistant

The Tester's Assistant is currently limited by the size of the code it is analyzing; the largest body of code that has been successfully statically analyzed is 2000–3000 lines. Many interesting programs are significantly larger than this; for example, sendmail is 69,000 lines distributed across many source files. Two changes will address this problem: to allow multiple source files to be processed, and to process one function at a time, scheduling function processing via a call graph of the whole program to reduce the amount of information that needs to be stored at any single point in time.

One difficulty with data-flow analysis of multiple source files maintaining a persistent multi-file data-flow graph between invocations of the Tester's Assistant. With this innovation, it becomes realistic to do exact analysis of library calls and other system code. A database of behavior of standard functions enables exact analysis without the time-intensive process of manually specifying data-flow behavior for every library call in a large program.

Eventually, for a tool such as the Tester's Assistant to become more widely used, it should include a graphical user interface. Other static analysis tools have graphical interfaces, so what such a graphical interface would look like has been established. The display of the source text of the program would be configured such that code represented by significant structures in the data-flow graph could be highlighted, and thereby graphically selected as slicing criteria. Slice output could also be displayed in a similar fashion, with code included in the slice being displayed in a different color than is surrounding code not in the slice. Similarly, coverage results could be viewed graphically. Early development on the Tester's Assistant has concentrated on slicing, testing, and other algorithmic design issues, saving the user interface for future work.

Farther in the future it is desirable to link the Tester's Assistant with other testing tools. The Tester's Assistant is designed to fill a few roles in testing, such as goal-directed coverage criteria and oracles, and by linkage of testing results to formal specifications. Other aspects of testing, such as test data generation, are not addressed, so additional tools are needed to complete the Tester's Assistant.

Chapter 9

Results and Conclusions

9.1 Summary

This dissertation has described property-based testing, a new approach to detecting flaws (or the absence thereof) in software, with some assurance as to the correctness of the result of the test.

Property-based testing derives a structural test of a program from the specification of a generic property. A TASPEC specification corresponds to specific source code structures. Code locations which influence the property are automatically identified, and thoroughly tested. Thereby, confidence in the model of reasoning used in the generation of the specification is translated into confidence of the correctness of source code with respect to that property.

TASPEC provides a layer of low-level specification primitives with which properties can be specified. The design of TASPEC was intentionally kept simple to enable automatic translation from TASPEC into program slicing criteria and execution monitors. TASPEC is an expressive language which is capable of describing generic flaws. The semantic primitive is a predicate, denoted by a predicate name and zero or more parameters. Predicates can be connected via formulas containing common logical and temporal connectives. Predicates are "created" by the tagging of code locations or by combination of other predicates. The information that locations are tied to specific predicates is used both in the static and the dynamic analysis of the program.

Statically, the information is used for program slicing. Dynamically, the information is used for execution monitoring.

Program slicing finds a semantically closed subset of a program with respect to a given property. A property defines code locations (and relations between locations) which are significant. The slice finds all portions of the program which affect the values of important variables at these locations. A program slice is useful in both informal and formal ways. Informally, a slice can be inspected in an effort to spot logic or other errors. Formally, the slice can be further analyzed with algorithms which may have been too expensive to have been run on the program as a whole.

To calculate a program slice, a complete data and control dependency graph is needed for the program. For C programs, program slicers have not been developed in the past because of several technical difficulties, such as the issue of ill-behaved pointers. The basis for dealing with ill-behaved programs is that potentially ill-behaved programs are easy to detect, by reason of the intractability of program analysis for that program. For most programs, this is enough of a negative result to necessitate other, more rigorous analysis or re-implementation.

The iterative contexts coverage metric is used as the measure of assurance. A single iterative context specifies a path which must be executed for that context to be covered. Path-based coverage is, in general, inefficient, but when the paths in a program slice only are considered, the number of potential paths radically decreases, assuming that an assumption about loops is made. Additionally, by basing the paths upon data dependencies, loops which have non-recursive data-flow graphs do not cause unbounded path sets.

The Tester's Assistant is a prototype system built to demonstrate the practicality and effectiveness of property-based testing. The Tester's Assistant can analyze C programs and produce slices of these programs. It generates a new version of the target program with added code that monitors program execution and gathers information related to test coverage. Analysis of several Ultrix programs using the Tester's Assistant shows that static analysis and slicing are practical and effective for finding a small subset of a program relative to a TASPEC specification.

9.2 Perspective

Property-based testing is a good technique for addressing the types of security flaws presented in this dissertation. The question is, for what other types of programs and flaws is property-based testing appropriate?

TASPEC can express a wide range of program properties but cannot express program-specific properties as easily as generic properties, because of the inability to target specific points in the program with location specifiers. TASPEC is also ill-suited for expressing structural flaws of programs, such as function calls with the wrong number of parameters. The handling of program-specific properties can be improved by generalizing location specifiers, but structural flaws can be handled more effectively be handled by tools such as lint.

The iterative contexts coverage metric is stronger than other related metrics, especially when additional partition information is added. The format of and technique for generating the additional partition information is future research. Iterative contexts can be generated for specific points in the program, unlike other metrics. If only a part of a program is of interest to the tester, then the number of contexts required by iterative contexts is much less than that required by other metrics. Therefore, iterative contexts is preferable for security testing, because typically only a small portion of a program is security-related.

Property-based testing is a tool-driven technology. Calculating correctness of properties and iterative context coverage for a large program is not trivial, and requires automated support. Because this support does not exist at this time, it is impossible to evaluate quantitatively the effectiveness of property-based testing and related testing techniques for general-purpose testing. A side effect of the requirement for automated support is the requirement for language-specific support. The Tester's Assistant analyzes C programs, and a significant amount of front-end support would be necessary to allow programs written in other languages to be analyzed.

Property-based testing is primarily useful for testing programs for generic flaws, i.e., security flaws, after the programs have been debugged with classic methods. Using property-based testing, a human tester can effectively focus resources on the most important properties of the program, and validate the program with respect to those properties.

9.3 Future work

The TASPEC-based execution monitor and the iterative contexts coverage analyzer have not been implemented. In order to fully evaluate property-based testing, data must be collected from complete property-based tests. The most important piece of future work is to finish the implementation and to collect empirical data on the property-based testing process. Once this evaluation has been completed, the extensions to property-based testing described in the rest of this section can be considered.

The work on property-based testing presented here is only part of a complete testing and validation system. The current work does not cover several areas, such as reasoning about the specification model, test data generation, and hierarchical construction of tests using the module structure of the program; further research in property-based testing is needed to address these issues.

More specifically, these are areas in which more research is needed:

- The Tester's Assistant internal form is specialized and somewhat language-specific. In order to make the tool set more easily adjustable to other programming languages, the internal form needs to be re-designed with simpler (and fewer) constructors. This problem was caused by the necessity of performing live assignment propagation in C; moving this phase of the graph construction to *lido* will enable the data-flow graph to be language-independent. New languages could then be added to the capability of the Tester's Assistant by implementing new construction and output routines in ELI.

- A more concrete representation of program paths can provide a broader base of path coverage metrics. A calculus could be developed to do path computations, providing reasoning abilities about the relationship of program traces, expressed as paths. One benefit of such a calculus would be that coverage contexts could be displayed in a more lucid fashion, enabling the human tester to more easily interpret and act on the results of the coverage metric [40].

- Techniques for initial test data generation have not been specified. This lack is not necessarily a weakness in property-based testing: initial data can be provided by the developers of the software. However, information expressed in the specification of properties (or in program specifications if available) could be used to generate initial test data; this information is otherwise ignored. TASPEC specifications, then, should be examined for its capabilities for initial test data generation.

- A facility to generate auxiliary test data based upon unexecuted portions of the program needs to be developed. The path coverage metrics reveal program paths and sub-paths which have not been executed for any data in the test suite. The assumption made so far is that some mechanism exists for finding test data for these paths. Symbolic evaluation [5] has shown promise in addressing gaps in coverage [11, 35].

- TASPEC specifications are very detailed; the language is designed for the specific correlation between events and code, and matching these with automatic testing methods. Specifying high-level abstract constructs is possible in TASPEC. However, more developed languages with higher-level constructs, such as Z, are the standard specification languages in industry. Therefore, it would be beneficial to translate the more widely-used specification languages to the TASPEC language so the specifications can be used in property-based testing. For concrete Z specifications, Helmke [26] has shown that semi-automatic translation from Z to TASPEC is possible.

- Data-flow testing is not the only approach to program testing. Mutation testing is one technique which approaches testing from a different perspective, by requiring coverage of only local structural features of the program (see Section 3.5.) An interesting question is "What adjustments to mutation testing are necessary to be able to provide results with respect to a property?" With a general internal program representation, other methods of test data generation and calculation of test completeness could be provided to the user of the Tester's Assistant.

- The examples in this work have come from the area of UNIX security, although in general TASPEC allows any arbitrary specification model to be expressed. Work is currently ongoing to specify safety properties and form corresponding property-based tests. TASPEC can generate property specifications for well-behavedness properties. Generalized program specifications have not been attempted. Completing property-based tests of all these types of properties also is future work.
- The Tester's Assistant would be more attractive and usable with a graphical user interface. The interface could show program slices, specification objects, and test data, and their relationship both statically and dynamically. This interface would help the human tester perform property-based testing more efficiently.
- Monolithic testing of a program is not an efficient way to obtain a validation. Rather, unit testing of the lower-level modules of a program should be done first, and only when these routines are validated does the system-level validation proceed. With the capability of property-based testing to specify abstract properties related to a location such as a function call, a promising avenue of research is to use property-based testing to establish abstract properties via a unit test of a function, and then use the abstract properties attached to function call of that function in the system test.

9.4 Conclusion

The goal of the research is to provide a practical method for validating that a program does not possess generic, specified flaws. The approach is to develop a program analysis and testing methodology, property-based testing, that can thoroughly test a program with respect to specifications of these flaws.

Specifications of generic properties in the TASPEC language drive the testing process, controlling how the other techniques are applied. Static analysis and program slicing are the most important techniques, as they reduce the problem size for other aspects of the analysis. Definition-use test coverage metrics provide completeness measures.

Unlike previous testing techniques that provided results exclusively with respect to specifications or code, but not both, property-based testing directly ties testing results to a specified property. Results from the testing process are in the form of `coverage` with respect to a `property`.

Property-based testing introduces a testing mechanism that focuses testing and analysis on satisfying the most important properties of a program. Property-based testing is a step towards making validation results from testing approach the validation results attainable from verification.

Appendix A

TASPEC Grammar

The grammar is expressed in the *con* language, the ELI [22] language for expressing concrete grammars. In *con*, literals are enclosed in single quotes. Alternate phrases for the same non-terminal are separated by "/". A "." indicates the end of the list of phrases for the given non-terminal.

```
varexpr:
   addr_op varexpr /
   identifier '[' varexpr ']' /
   identifier '.' identifier /
   identifier '->' identifier /
   variable .

addr_op: '*' / '&' .

variable: identifier / string_literal / integer_constant .

assertion: identifier '(' arglist ')' .

arglist: args / .

args: varexpr ',' args / varexpr .

varexpr: 'sizeof' '(' identifier ')' .

tempexpr:
   logicalexpr 'before' logicalexpr /
```

```
    logicalexpr 'until' logicalexpr /
    logicalexpr 'eventually' logicalexpr /
    logicalexpr .

logicalexpr:
    logicalexpr 'and' logicalexpr /
    logicalexpr 'or' logicalexpr /
    logicalexpr 'implies' logicalexpr /
    'not' logicalexpr /
    '(' logicalexpr ')' /
    relexpr .

relexpr: varexpr relop varexpr / assertion.

relop: '=' / '<' / '>' / '<>' .

location: 'funcall' identifier funct_prototype result_opt condition /
          'variable' varexpr result_opt condition /
  'decl' varexpr condition /
          'assign' varexpr result_opt condition .

result_opt: 'result' identifier / .

funct_prototype: '(' identifier_list ')' / .

condition: 'if' tempexpr / .

state_change: assert-or-retract iterator_opt identifier
                                    '(' paralist ')' .

iterator_opt: iterator / .

iterator: '(' identifier '=' variable ',' variable ')' .

assert-or-retract: 'assert' / 'retract'.

paralist: variable ',' paralist / .

specification:
  location '{' event_list '}' /
  event .
```

```
event: state_change ; / predicate ; .

predicate: tempexpr event_list_opt .

event_list_opt: '{' event_list '}' / .
```

Appendix B

Interactive Session with the Slicer

The following is a typescript of a session with the Tester's Assistant. The program used in this run is the Minix [62] login program. The program has already been run through the C pre-processor. Several interesting library functions have been specified from the command line as the Tester's Assistant program `ta.exe` is invoked. Several indicators as to the size of the graph are printed as analysis is complete. At the ==> prompt, the Tester's Assistant is waiting for user input.

The command script that was executed was:

`(slice)==>prog`	Display the whole program
`(prog)==>slice`	Return to the slice mode of display
`(slice)==>func chown`	Slice on chown (returns slice # 0)
`(slice)==>func setuid`	Slice on setuid (returns slice # 1)
`(slice)==>summary`	Show the slices in memory
`(slice)==>intersect 0 1`	Intersect slices # 0 and # 1 (returns slice # 2)
`(slice)==>summary`	
`(slice)==>union 0 1`	Union slices # 0 and # 1 (returns slice # 3)
`(slice)==>summary`	
`(slice)==>rerun 2`	Erroneous input
`(slice)==>replay 2`	Show slice # 2 again
`(slice)==>inst`	Switch to instrumentation display mode
`(inst) ==>replay 1`	Show slice # 1 with instrumentation
`(inst) ==>quit`	

The log is on the following pages. At several points in the log, the whole program is displayed; because the program was pre-processed, a large number of declarations from header files appear at the beginning of the program. To improve the readability of the log, these declarations are removed.

Three seconds elapsed between the invocation of the Tester's Assistant and the appearance of the command prompt (on a Sparc 20). After this initial analysis time, production of slices was instantaneous.

```
% ta.exe -lf strcpy/OI -lf strcat/OI -lf strrchr/II -lf strncpy/OII
-lf chown/III login.cpp
Times through the inner loop: 1069
Number of nodes: 747
    Usages    Assigns    Paras     Decls     Calls
       167         43        94       183        72
Edge index: 3228

(slice)==>prog

/* Header declarations removed here */

 void wtmp ( line , user )
char *  line ;
char *  user ; {
   struct utmp entry , oldent ;
   char *  blank = "              " ;
   register int fd ;
   register char *  sp ;
   int lineno ;
   extern long time (  ) ;
   sp = strrchr ( line , '/' ) ;
   if ( sp == 0 )
   sp = line ; else
   sp ++ ;
   if ( ( fd = open ( UTMP , 0 ) ) < 0 )
   return  ;
   lineno = 0 ;
   while ( read ( fd , ( char *  ) & oldent , sizeof ( struct
utmp  ) ) == sizeof ( struct utmp  ) )
   {
      if ( oldent . ut_pid == getpid ( ) )
      break ;
      lineno ++ ; }
   lineno *= sizeof ( struct utmp  ) ;
   if ( lseek ( fd , ( long  ) lineno , 0 ) >= 0 )
   {
      read ( fd , ( char *  ) & oldent , sizeof ( struct
utmp  ) ) ; }
   close ( fd ) ;
   strncpy ( entry . ut_user , blank , ( entry . ut_user ) ) ;
   strncpy ( entry . ut_id , blank , ( entry . ut_id ) ) ;
```

```
    strncpy ( entry . ut_line , blank , ( entry . ut_line ) ) ;
    strncpy ( entry . ut_user , user , ( entry . ut_user ) ) ;
    strncpy ( entry . ut_id , oldent . ut_id , ( entry . ut_id ) ) ;
    strncpy ( entry . ut_line , sp , ( entry . ut_line ) ) ;
    entry . ut_pid = oldent . ut_pid ;
    entry . ut_type = USER_PROCESS ;
    time ( & ( entry . ut_time ) ) ;
    if ( ( fd = open ( WTMP , 1 ) ) > 0 )
    {
       if ( lseek ( fd , 0 , 2 ) >= 0 )
       {
          write ( fd , ( char *  ) & entry ,
                           sizeof ( struct utmp  ) ) ; }
       close ( fd ) ; }
    if ( ( fd = open ( UTMP , 1 ) ) > 0 )
    {
       if ( lseek ( fd , ( long  ) lineno , 0 ) >= 0 )
       {
          write ( fd , ( char *  ) & entry ,
                           sizeof ( struct utmp  ) ) ; }
       close ( fd ) ; } }
 void show_file ( nam )
char *  nam ; {
   register int fd , len ;
   char buf [ 80 ] ;
   if ( ( fd = open ( nam , 0 ) ) > 0 )
   {
      len = 1 ;
      while ( len > 0 )
      {
         len = read ( fd , buf , 80 ) ;
         write ( 1 , buf , len ) ; }
      close ( fd ) ; } }
 int main ( argc , argv )
int argc ;
char *  argv [  ] ; {
   char name [ 30 ] ;
   char password [ 30 ] ;
   char ttyname [ 16 ] ;
   int bad , n , ttynr , ap ;
   struct sgttyb args ;
   struct passwd *  pwd ;
   struct stat statbuf ;
   char * bp , * argx [ 8 ] ;
   char * sh = "/bin/sh" ;
   char * sh2 = "/usr/bin/sh" ;
   if ( ioctl ( 0 , ( 1073741824 | ( ( sizeof ( struct sgttyb )
& 255 ) << 16 ) | ( 't' << 8 ) | 8 ) , & args ) < 0 )
   exit ( 1 ) ;
   fstat ( 0 , & statbuf ) ;
   ttynr = statbuf . st_rdev & 255 ;
   if ( ttynr == 0 )
   strcpy ( ttyname , "tty0" ) ; else
   {
      strcpy ( ttyname , "tty?" ) ;
      ttyname [ 3 ] = '0' + ttynr ; }
   for ( ; ; )
   {
      bad = 0 ;
      if ( argc > 1 )
      {
         strcpy ( name , argv [ 1 ] ) ;
         argc = 1 ; } else
      {
         do
         {
            write ( 1 , "login: " , 7 ) ;
            n = read ( 0 , name , 30 ) ; } while ( n < 2 ) ;
         name [ n - 1 ] = 0 ; }
      if ( ( pwd = getpwnam ( name ) ) == ( struct passwd * ) 0 )
      bad ++ ;
      if ( bad || strlen ( pwd -> pw_passwd ) != 0 )
      {
         args . sg_flags &= ~ 8 ;
         ioctl ( 0 , ( -2147483648 | ( ( sizeof ( struct sgttyb )
& 255 ) << 16 ) | ( 't' << 8 ) | 9 ) , & args ) ;
         write ( 1 , "Password: " , 10 ) ;
         time_out = 0 ;
         signal ( 14 , Time_out ) ;
         alarm ( 30 ) ;
         n = read ( 0 , password , 30 ) ;
         alarm ( 0 ) ;
         if ( time_out )
         {
            n = 1 ;
            bad ++ ; }
         password [ n - 1 ] = 0 ;
         write ( 1 , "\n" , 1 ) ;
         args . sg_flags |= 8 ;
         ioctl ( 0 , ( -2147483648 | ( ( sizeof ( struct sgttyb  )
& 255 ) << 16 ) | ( 't' << 8 ) | 9 ) , & args ) ;
         if ( bad && crypt ( password , "aaaa" ) || strcmp (
pwd -> pw_passwd , crypt ( password , pwd -> pw_passwd ) ) )
         {
            write ( 1 , "Login incorrect\n" , 16 ) ;
            continue ; } }
      if ( access ( "/etc/nologin" , 0 ) == 0 && strcmp ( name ,
"root" ) != 0 )
      {
         write ( 1 , "System going down\n\n" , 19 ) ;
         continue ; }
      wtmp ( ttyname , name ) ;
      ap = 0 ;
      argx [ ap ++ ] = "-" ;
      if ( pwd -> pw_shell [ 0 ] )
      {
         sh = pwd -> pw_shell ;
         bp = sh ;
         while ( * bp )
         {
            while ( * bp && * bp != ' ' && * bp != '\t' )
            bp ++ ;
            if ( * bp == ' ' || * bp == '\t' )
            {
               * bp ++ = '\0' ;
               argx [ ap ++ ] = bp ; } } } else
      argx [ ap ] = ( char *  ) 0 ;
      strcpy ( user , "USER=" ) ;
      strcat ( user , name ) ;
      strcpy ( logname , "LOGNAME=" ) ;
      strcat ( logname , name ) ;
      strcpy ( home , "HOME=" ) ;
      strcat ( home , pwd -> pw_dir ) ;
      strcpy ( shell , "SHELL=" ) ;
      strcat ( shell , sh ) ;
      chdir ( pwd -> pw_dir ) ;
      for ( n = 1 ; n <= _NSIG ; ++ n )
      signal ( n , ( void *  (  ) ) 0 ) ;
      show_file ( "/etc/motd" ) ;
      strcpy ( name , "/dev/" ) ;
      strcat ( name , ttyname ) ;
      chown ( name , pwd -> pw_uid , pwd -> pw_gid ) ;
      setgid ( pwd -> pw_gid ) ;
      setuid ( pwd -> pw_uid ) ;
      execve ( sh , argx , env ) ;
      execve ( sh2 , argx , env ) ;
      write ( 1 , "exec failure\n" , 13 ) ;
      exit ( 1 ) ; } }
 void Time_out (  )  {
   time_out = 1 ; }
(prog) ==>slice
(slice)==>func chown
Slice #0 : (Backward slice for function call chown slice formatted)
Slice size: 62 nodes

int time_out ;
 int main ( argc , argv )
int argc ;
char *  argv [  ] ; {
   char name [ 30 ] ;
   char password [ 30 ] ;
   char ttyname [ 16 ] ;
   int bad , n , ttynr , ap ;
   struct passwd *  pwd ;
   for (  ;  ;  )
   {
      bad = 0 ;
      if ( ( pwd = getpwnam ( name ) ) == ( struct passwd *  ) 0 )
      bad ++ ;
      if ( bad || strlen ( pwd -> pw_passwd ) != 0 )
      {
         time_out = 0 ;
         read ( 0 , password , 30 ) ;
         if ( time_out )
         {
            bad ++ ; }
         password [ n - 1 ] = 0 ;
         if ( bad && crypt ( password , "aaaa" ) || strcmp ( pwd ->
pw_passwd , crypt ( password , pwd -> pw_passwd ) ) )
         {
            continue ; } }
      if ( access ( "/etc/nologin" , 0 ) == 0 && strcmp ( name ,
"root" ) != 0 )
      {
         continue ; }
      chdir ( pwd -> pw_dir ) ;
      strcat ( name , ttyname ) ;
      chown ( name , pwd -> pw_uid , pwd -> pw_gid ) ;       } }
(slice)==>func setuid
Slice #1 : (Backward slice for function call setuid slice formatted)
Slice size: 27 nodes

int time_out ;
 int main ( argc , argv )
int argc ;
char *  argv [  ] ; {
   int bad , n , ttynr , ap ;
   for (  ;  ;  )
   {
      bad = 0 ;
      if ( ( pwd = getpwnam ( name ) ) == ( struct passwd *  ) 0 )
      bad ++ ;
      if ( bad || strlen ( pwd -> pw_passwd ) != 0 )
      {
         time_out = 0 ;
         if ( time_out )
         {
            bad ++ ; }
         if ( bad && crypt ( password , "aaaa" ) || strcmp (
pwd -> pw_passwd , crypt ( password , pwd -> pw_passwd ) ) )
         {
            continue ; } }
      if ( access ( "/etc/nologin" , 0 ) == 0 && strcmp ( name ,
"root" ) != 0 )
      {
         continue ; }
      setuid ( pwd -> pw_uid ) ;     } }
(slice)==>summary
Slice #0 : (Backward slice for function call chown slice formatted)
Slice #1 : (Backward slice for function call setuid slice formatted)
```

```
(slice)==>intersect 0 1
Slice #2 is the intersection of slices 0 and 1

int time_out ;
 int main ( argc , argv )
int argc ;
char *  argv [  ] ; {
   int bad , n , ttynr , ap ;
   for (  ;  ;  )
   {
      bad = 0 ;
      if ( ( pwd = getpwnam ( name ) ) == ( struct passwd *  ) 0 )
      bad ++ ;
      if ( bad || strlen ( pwd -> pw_passwd ) != 0 )
      {
         time_out = 0 ;
         if ( time_out )
         {
            bad ++ ; }
         if ( bad && crypt ( password , "aaaa" ) || strcmp (
pwd -> pw_passwd , crypt ( password , pwd -> pw_passwd ) ) )
         {
            continue ; } }
      if ( access ( "/etc/nologin" , 0 ) == 0 && strcmp ( name ,
"root" ) != 0 )
      {
         continue ; }                          } }
(slice)==>summary
Slice #0 : (Backward slice for function call chown slice formatted)
Slice #1 : (Backward slice for function call setuid slice formatted)
Slice #2 is the intersection of slices 0 and 1
(slice)==>union 0 1
Slice #3 is the union of slices 0 and 1

int time_out ;
 int main ( argc , argv )
int argc ;
char *  argv [  ] ; {
   char name [ 30 ] ;
   char password [ 30 ] ;
   char ttyname [ 16 ] ;
   int bad , n , ttynr , ap ;
   struct passwd *  pwd ;
   for (  ;  ;  )
   {
      bad = 0 ;
      if ( ( pwd = getpwnam ( name ) ) == ( struct passwd *  ) 0 )
      bad ++ ;
      if ( bad || strlen ( pwd -> pw_passwd ) != 0 )
      {
         time_out = 0 ;
         read ( 0 , password , 30 ) ;
         if ( time_out )
         {
            bad ++ ; }
         password [ n - 1 ] = 0 ;
         if ( bad && crypt ( password , "aaaa" ) || strcmp (
 pwd -> pw_passwd , crypt ( password , pwd -> pw_passwd ) ) )
         {
            continue ; } }
      if ( access ( "/etc/nologin" , 0 ) == 0 && strcmp ( name ,
"root" ) != 0 )
      {
         continue ; }
      chdir ( pwd -> pw_dir ) ;
      strcat ( name , ttyname ) ;
      chown ( name , pwd -> pw_uid , pwd -> pw_gid ) ;
      setuid ( pwd -> pw_uid ) ;     } }
(slice)==>summary
Slice #0 : (Backward slice for function call chown slice formatted)
Slice #1 : (Backward slice for function call setuid slice formatted)
Slice #2 is the intersection of slices 0 and 1
Slice #3 is the union of slices 0 and 1
(slice)==>rerun 2
enter a valid command
(slice)==>replay 2
Slice #2 is the intersection of slices 0 and 1

int time_out ;
 int main ( argc , argv )
int argc ;
char *  argv [  ] ; {
   int bad , n , ttynr , ap ;
   for (  ;  ;  )
```

```
  {
     bad = 0 ;
     if ( ( pwd = getpwnam ( name ) ) == ( struct passwd *  ) 0 )
     bad ++ ;
     if ( bad || strlen ( pwd -> pw_passwd ) != 0 )
     {
        time_out = 0 ;
        if ( time_out )
        {
           bad ++ ; }
        if ( bad && crypt ( password , "aaaa" ) || strcmp (
pwd -> pw_passwd , crypt ( password , pwd -> pw_passwd ) ) )
        {
           continue ; } }
     if ( access ( "/etc/nologin" , 0 ) == 0 && strcmp ( name ,
"root" ) != 0 )
     {
        continue ; }                          } }
(slice)==>inst
(inst) ==>replay 1
Slice #1 : (Backward slice for function call setuid slice formatted)
/* The next four lines set up the recording structure for the trace
   Any call to TADFC(x) is added during instrumentation */

#define TADFC(x) path[path_size++] = x
#define MAXDFPATH 50000
int path[MAXDFPATH];
int path_size = 0;

/* Header declarations removed here */

 void wtmp ( line , user )
char *  line ;
char *  user ; {
   struct utmp entry , oldent ;
   char *  blank = "                 " ;
   register int fd ;
   register char *  sp ;
   int lineno ;
   extern long time (  ) ;
   sp = strrchr ( line , '/' ) ;
   if ( sp == 0 )
   sp = line ; else
   sp ++ ;
   if ( ( fd = open ( UTMP , 0 ) ) < 0 )
   return  ;
   lineno = 0 ;
   while ( read ( fd , ( char *  ) & oldent , sizeof ( struct
utmp  ) ) == sizeof ( struct utmp  ) )
   {
      if ( oldent . ut_pid == getpid ( ) )
      break ;
      lineno ++ ; }
   lineno *= sizeof ( struct utmp  ) ;
   if ( lseek ( fd , ( long  ) lineno , 0 ) >= 0 )
   {
      read ( fd , ( char *  ) & oldent , sizeof ( struct utmp  ) ) ; }
   close ( fd ) ;
   strncpy ( entry . ut_user , blank , ( entry . ut_user ) ) ;
   strncpy ( entry . ut_id , blank , ( entry . ut_id ) ) ;
   strncpy ( entry . ut_line , blank , ( entry . ut_line ) ) ;
   strncpy ( entry . ut_user , user , ( entry . ut_user ) ) ;
   strncpy ( entry . ut_id , oldent . ut_id , ( entry . ut_id ) ) ;
   strncpy ( entry . ut_line , sp , ( entry . ut_line ) ) ;
   entry . ut_pid = oldent . ut_pid ;
   entry . ut_type = USER_PROCESS ;
   time ( & ( entry . ut_time ) ) ;
   if ( ( fd = open ( WTMP , 1 ) ) > 0 )
   {
      if ( lseek ( fd , 0 , 2 ) >= 0 )
      {
         write ( fd , ( char *  ) & entry ,
                        sizeof ( struct utmp  ) ) ; }
      close ( fd ) ; }
   if ( ( fd = open ( UTMP , 1 ) ) > 0 )
   {
      if ( lseek ( fd , ( long  ) lineno , 0 ) >= 0 )
      {
         write ( fd , ( char *  ) & entry ,
                        sizeof ( struct utmp  ) ) ; }
      close ( fd ) ; } }
void show_file ( nam )
char *  nam ; {
  register int fd , len ;
```

```
   char buf [ 80 ] ;
   if ( ( fd = open ( nam , 0 ) ) > 0 )
   {
      len = 1 ;
      while ( len > 0 )
      {
         len = read ( fd , buf , 80 ) ;
         write ( 1 , buf , len ) ; }
      close ( fd ) ; } }
 int main ( argc , argv )
int argc ;
char *  argv [  ] ; {
   char name [ 30 ] ;
   char password [ 30 ] ;
   char ttyname [ 16 ] ;
   int bad , n , ttynr , ap ;
   struct sgttyb args ;
   struct passwd *  pwd ;
   struct stat statbuf ;
   char *  bp , *  argx [ 8 ] ;
   char *  sh = "/bin/sh" ;
   char *  sh2 = "/usr/bin/sh" ;
   if ( ioctl ( 0 , ( 1073741824 | ( ( sizeof ( struct sgttyb  )
& 255 ) << 16 ) | ( 't' << 8 ) | 8 ) , & args ) < 0 )
   exit ( 1 ) ;
   fstat ( 0 , & statbuf ) ;
   ttynr = statbuf . st_rdev & 255 ;
   if ( ttynr == 0 )
   strcpy ( ttyname , "tty0" ) ; else
   {
      strcpy ( ttyname , "tty?" ) ;
      ttyname [ 3 ] = '0' + ttynr ; }
   {TADFC(717);for (  ;  ;  ) {
   {
      {TADFC(442);bad = 0;} ;
      if ( argc > 1 )
      {
         strcpy ( name , argv [ 1 ] ) ;
         argc = 1 ; } else
      {
         do
         {
                write ( 1 , "login: " , 7 ) ;
                n = read ( 0 , name , 30 ) ; } while ( n < 2 ) ;
            name [ n - 1 ] = 0 ; }
        {TADFC(476);if ( ( pwd = (TADFC(472),getpwnam ( name )) )
== ( struct passwd *  ) 0 ) {
        (TADFC(475),(TADFC(474),bad) ++) ;};}
        {TADFC(549);if ( (TADFC(477),bad) || (TADFC(482),strlen (
pwd -> pw_passwd )) != 0 ) {
        {
            args . sg_flags &= ~ 8 ;
            ioctl ( 0 , ( -2147483648 | ( ( sizeof ( struct sgttyb  )
& 255 ) << 16 ) | ( 't' << 8 ) | 9 ) , & args ) ;
            write ( 1 , "Password: " , 10 ) ;
            {TADFC(493);time_out = 0;} ;
            signal ( 14 , Time_out ) ;
            alarm ( 30 ) ;
            n = read ( 0 , password , 30 ) ;
            alarm ( 0 ) ;
            {TADFC(513);if ( (TADFC(508),time_out) ) {
            {
                n = 1 ;
                (TADFC(512),(TADFC(511),bad) ++) ; }};}
            password [ n - 1 ] = 0 ;
            write ( 1 , "\n" , 1 ) ;
            args . sg_flags |= 8 ;
            ioctl ( 0 , ( -2147483648 | ( ( sizeof ( struct sgttyb  )
& 255 ) << 16 ) | ( 't' << 8 ) | 9 ) , & args ) ;
            {TADFC(548);if ( (TADFC(527),bad) && (TADFC(531),crypt (
password , "aaaa" )) || (TADFC(544),strcmp ( pwd -> pw_passwd ,
crypt ( password , pwd -> pw_passwd ) )) ) {
            {
                write ( 1 , "Login incorrect\n" , 16 ) ;
                {TADFC(547);continue ;;} }};} }};}
        {TADFC(559);if ( (TADFC(551),access ( "/etc/nologin" , 0 ))
 == 0 && (TADFC(555),strcmp ( name , "root" )) != 0 ) {
        {
            write ( 1 , "System going down\n\n" , 19 ) ;
            {TADFC(558);continue ;;} }};}
        wtmp ( ttyname , name ) ;
        ap = 0 ;
        argx [ ap ++ ] = "-" ;
        if ( pwd -> pw_shell [ 0 ] )
```

```
      {
         sh = pwd -> pw_shell ;
         bp = sh ;
         while ( * bp )
         {
            while ( * bp && * bp != ' ' && * bp != '\t' )
            bp ++ ;
            if ( * bp == ' ' || * bp == '\t' )
            {
               * bp ++ = '\0' ;
               argx [ ap ++ ] = bp ; } } } else
      argx [ ap ] = ( char *  ) 0 ;
      strcpy ( user , "USER=" ) ;
      strcat ( user , name ) ;
      strcpy ( logname , "LOGNAME=" ) ;
      strcat ( logname , name ) ;
      strcpy ( home , "HOME=" ) ;
      strcat ( home , pwd -> pw_dir ) ;
      strcpy ( shell , "SHELL=" ) ;
      strcat ( shell , sh ) ;
      chdir ( pwd -> pw_dir ) ;
      for ( n = 1 ; n <= _NSIG ; ++ n )
      signal ( n , ( void *  (  ) ) 0 ) ;
      show_file ( "/etc/motd" ) ;
      strcpy ( name , "/dev/" ) ;
      strcat ( name , ttyname ) ;
      chown ( name , pwd -> pw_uid , pwd -> pw_gid ) ;
      setgid ( pwd -> pw_gid ) ;
      (TADFC(696),setuid ( pwd -> pw_uid )) ;
      execve ( sh , argx , env ) ;
      execve ( sh2 , argx , env ) ;
      write ( 1 , "exec failure\n" , 13 ) ;
      exit ( 1 ) ; }};} }
 void Time_out (  )  {
   time_out = 1 ; }
(inst) ==>quit
%
```

Appendix C

Abstract Syntax for C

The abstract syntax for C used in the Tester's Assistant is derived from the ANSI C standard language definition (ANSI/ISO 9899-1990) [32]. In the derivation, some non-terminals are merged or reduced. For example, the `statement`, `statement_list`, and `while_statement` non-terminals are replaced with the single `Stmt` non-terminal. The final abstract syntax is in ELI's *lido* language [22].

```
RULE rule_001:  constant ::=  floating_constant

RULE rule_002:  constant ::=  integer_constant

RULE rule_003:  constant ::=  character_constant

RULE rule_004:  enumeration_constant ::=  identifier

RULE rule_005:  StringSeq ::=  string_literal

RULE rule_006:  StringSeq ::=  StringSeq string_literal

RULE rule_007:  LabelDef ::=  identifier

RULE rule_008:  LabelDef ::=  typedef_name

RULE rule_009:  LabelDef ::=  OrdinaryIdUse

RULE rule_010:  LabelUse ::=  identifier
RULE rule_011:  MemberIdDef ::=  identifier

RULE rule_012:  MemberIdUse ::=  identifier

RULE rule_013:  IdDef ::=  OrdinaryIdDef

RULE rule_014:  Para ::=  OrdinaryIdDef

RULE rule_015:  IdUse ::=  identifier

RULE rule_016:  IdUse ::=  OrdinaryIdUse

RULE rule_017:  Expr ::=  IdUse

RULE rule_018:  Expr ::=  constant

RULE rule_019:  Expr ::=  StringSeq

RULE rule_020:  Expr ::=  FExpr

RULE rule_021:  Expr ::=  VExpr

RULE rule_022:  Expr ::=  Asgn

RULE rule_023:  VExpr ::=  Expr '[' Expr ']'

RULE rule_024:  VExpr ::=  Expr '.' MemberIdUse

RULE rule_025:  VExpr ::=  Expr '->' MemberIdUse

RULE rule_026:  Asgn ::=  Expr '++'

RULE rule_027:  Asgn ::=  Expr '--'

RULE rule_028:  FExpr ::=  Expr '(' Args ')'

RULE rule_029:  FExpr ::=  Expr '(' ')'

RULE rule_030:  Args ::=  Expr

RULE rule_031:  Args ::=  Args ',' Expr
RULE rule_032:  Expr ::=  UnOp Expr

RULE rule_033:  Expr ::=  AddrOp Expr

RULE rule_034:  Expr ::=  'sizeof' '(' type_name ')'

RULE rule_035:  Asgn ::=  '++' Expr

RULE rule_036:  Asgn ::=  '--' Expr

RULE rule_037:  UnOp ::=  '+'

RULE rule_038:  UnOp ::=  '-'

RULE rule_039:  UnOp ::=  '~'

RULE rule_040:  UnOp ::=  '!'

RULE rule_041:  AddrOp ::=  '&'

RULE rule_042:  AddrOp ::=  '*'

RULE rule_043:  Expr ::=  '(' type_name ')' Expr

RULE rule_044:  Expr ::=  Expr BinOp Expr

RULE rule_045:  BinOp ::=  '*'

RULE rule_046:  BinOp ::=  '/'

RULE rule_047:  BinOp ::=  '%'

RULE rule_048:  BinOp ::=  '+'

RULE rule_049:  BinOp ::=  '-'

RULE rule_050:  BinOp ::=  '<<'

RULE rule_051:  BinOp ::=  '>>'

RULE rule_052:  BinOp ::=  '<'
RULE rule_053:  BinOp ::=  '>'

RULE rule_054:  BinOp ::=  '<='

RULE rule_055:  BinOp ::=  '>='

RULE rule_056:  BinOp ::=  '=='

RULE rule_057:  BinOp ::=  '!='

RULE rule_058:  BinOp ::=  '&'

RULE rule_059:  BinOp ::=  '^'

RULE rule_060:  BinOp ::=  '|'

RULE rule_061:  BinOp ::=  '&&'

RULE rule_062:  BinOp ::=  '||'

RULE rule_063:  CExpr ::=  Expr '?' Expr ':' Expr

RULE rule_064:  Expr ::=  CExpr

RULE rule_065:  Asgn ::=  Expr SAsgnOp Expr

RULE rule_066:  Asgn ::=  Expr AsgnOp Expr

RULE rule_067:  AsgnOp ::=  '*='

RULE rule_068:  AsgnOp ::=  '/='

RULE rule_069:  AsgnOp ::=  '%='

RULE rule_070:  AsgnOp ::=  '+='

RULE rule_071:  AsgnOp ::=  '-='

RULE rule_072:  AsgnOp ::=  '<<='

RULE rule_073:  AsgnOp ::=  '>>='
RULE rule_074:  AsgnOp ::=  '&='

RULE rule_075:  AsgnOp ::=  '^='

RULE rule_076:  AsgnOp ::=  '|='

RULE rule_077:  SAsgnOp ::=  '='

RULE rule_078:  Expr ::=  Expr ',' Expr

RULE rule_079:  Decl ::=  Type Declarator ';'

RULE rule_080:  Type ::=  Type Type

RULE rule_081:  Declarator ::=  Declarator ',' Declarator

RULE rule_082:  Declarator ::=  Declarator '=' Init

RULE rule_083:  Type ::=  'typedef'

RULE rule_084:  Type ::=  'extern'

RULE rule_085:  Type ::=  'static'

RULE rule_086:  Type ::=  'auto'

RULE rule_087:  Type ::=  'register'

RULE rule_088:  Type ::=  'void'

RULE rule_089:  Type ::=  'float'

RULE rule_090:  Type ::=  struct_or_union_specifier

RULE rule_091:  Type ::=  enum_specifier

RULE rule_092:  Type ::=  typedef_name

RULE rule_093:  Type ::=  'char'

RULE rule_094:  Type ::=  'short'
RULE rule_095:  Type ::=  'int'

RULE rule_096:  Type ::=  'long'

RULE rule_097:  Type ::=  'double'

RULE rule_098:  Type ::=  'signed'

RULE rule_099:  Type ::=  'unsigned'

RULE rule_100:  struct_or_union_specifier ::=
                          struct_or_union TagDef_opt SBlock

RULE rule_101:  struct_or_union_specifier ::=  struct_or_union Tag

RULE rule_102:  SBlock ::=  '{' SDecl '}'

RULE rule_103:  struct_or_union ::=  'struct'

RULE rule_104:  struct_or_union ::=  'union'

RULE rule_105:  SDecl ::=  SDecl SDecl

RULE rule_106:  SDecl ::=  Type SDeclarator ';'

RULE rule_107:  SDeclarator ::=  SDeclarator ',' SDeclarator

RULE rule_108:  SDeclarator ::=  SDeclarator ':' Expr

RULE rule_109:  SDeclarator ::=  pointer SDeclarator

RULE rule_110:  SDeclarator ::=  MemberIdDef

RULE rule_111:  SDeclarator ::=  SDeclarator '[' constant_exp_opt ']'

RULE rule_112:  SDeclarator ::=  SDeclarator Paras

RULE rule_113:  enum_specifier ::=  'enum' TagDef_opt '{' Enum '}'

RULE rule_114:  enum_specifier ::=  'enum' TagUse
RULE rule_115:  Enum ::=  Enum ',' Enum

RULE rule_116:  Enum ::=  enumeration_constant

RULE rule_117:  Enum ::=  enumeration_constant '=' Expr

RULE rule_118:  TagDef ::=  identifier

RULE rule_119:  Tag ::=  identifier

RULE rule_120:  Decl ::=  struct_or_union TagDef ';'

RULE rule_121:  TagUse ::=  identifier

RULE rule_122:  Type ::=  'const'

RULE rule_123:  Type ::=  'volatile'

RULE rule_124:  Declarator ::=  pointer Declarator

RULE rule_125:  Declarator ::=  IdDef

RULE rule_126:  Declarator ::=  Declarator '[' constant_exp_opt ']'

RULE rule_127:  Declarator ::=  Declarator Paras

RULE rule_128:  Paras ::=  '(' Para ')'

RULE rule_129:  pointer ::=  '*' Type

RULE rule_130:  pointer ::=  '*' Type pointer

RULE rule_131:  Type ::=

RULE rule_132:  Para ::=  Para ',' Para

RULE rule_133:  Para ::=  Type Declarator

RULE rule_134:  Para ::=  Type TDeclarator

RULE rule_135:  type_name ::=  Type TDeclarator
RULE rule_136:  TDeclarator ::=  pointer

RULE rule_137:  TDeclarator ::=  pointer TDeclarator

RULE rule_138:  TDeclarator ::=  TDeclarator '[' constant_exp_opt ']'

RULE rule_139:  TDeclarator

RULE  rule_140:  TDeclarator ::=  '[' constant_exp_opt ']'

RULE rule_141:  TDeclarator ::=  '(' Para ')'

RULE rule_142:  Init ::=  Expr

RULE rule_143:  Init ::=  Init ',' Init

RULE rule_144:  Stmt ::=  LStmt

RULE rule_145:  Stmt ::=  Block

RULE rule_146:  Stmt ::=  CStmt

RULE rule_147:  Stmt ::=  JStmt

RULE rule_148:  LStmt ::=  LabelDef ':' Stmt

RULE rule_149:  LStmt ::=  'case' Expr ':' Stmt

RULE rule_150:  LStmt ::=  'default' ':' Stmt

RULE rule_151:  Block ::=  '{' Decl Stmt '}'

RULE rule_152:  Block ::=  '{' Stmt '}'

RULE rule_153:  Decl ::=  Decl Decl

RULE rule_154:  Stmt ::=  Stmt Stmt

RULE rule_155:  Stmt ::=  Expr ';'

RULE rule_156:  CStmt ::=  'if' '(' Expr ')' Stmt
RULE rule_157:  CStmt ::=  'if' '(' Expr ')' Stmt 'else' Stmt

RULE rule_158:  CStmt ::=  'switch' '(' Expr ')' Stmt

RULE rule_159:  CStmt ::=  'while' '(' Expr ')' Stmt

RULE rule_160:  CStmt ::=  'do' Stmt 'while' '(' Expr ')' ';'

RULE rule_161:  CStmt ::=  'for' '(' Expr ';' Expr ';' Expr ')' Stmt

RULE rule_162:  Expr ::=

RULE rule_163:  JStmt ::=  'goto' LabelUse ';'

RULE rule_164:  JStmt ::=  'continue' ';'

RULE rule_165:  JStmt ::=  'break' ';'

RULE rule_166:  JStmt ::=  'return' Expr ';'

RULE rule_167:  translation_unit ::=  Decl

RULE rule_168:  translation_unit ::=  translation_unit Decl

RULE rule_169:  Decl ::=  Decl_Head FDecl

RULE rule_170:  Decl ::=  Decl_Head Decl

RULE rule_171:  Decl_Head ::=

RULE rule_172:  FDecl ::=  Type Declarator Decl_opt Block

RULE rule_173:  FDecl ::=  Declarator Decl_opt Block

RULE rule_174:  TDeclarator ::=

RULE rule_175:  constant_exp_opt ::=

RULE rule_176:  constant_exp_opt ::=  Expr

RULE rule_177:  Decl_opt ::=  empty
RULE rule_178:  Decl_opt ::=  Decl

RULE rule_179:  SDeclarator ::=

RULE rule_180:  Para ::=

RULE rule_181:  Declarator ::=

RULE rule_182:  Stmt ::=

RULE rule_183:  TagDef_opt ::=

RULE rule_184:  TagDef_opt ::=  TagDef

RULE rule_185:  empty ::=

RULE rule_186:  root ::=  source

RULE rule_187:  source ::=

RULE rule_188:  source ::=  file

RULE rule_189:  file ::=  translation_unit

RULE rule_1:  Expr ::=  '(' Expr ')'
```

Bibliography

[1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison/Wesley Publishing Company, 1986.

[2] Derek Andrews and Darrel Ince. *Practical Formal Methods with VDM.* McGraw-Hill, 1991.

[3] Thomas Ball and Susan Horwitz. Slicing programs with arbitrary control flow. In *Automated and Algorithmic Debugging, First International Workshop '93 Proceedings*, pages 206–222, May 1993.

[4] Thomas J. Ball and James R. Larus. Optimally profiling and tracing programs. Technical Report CS-TR-91-1031, University of Wisconsin, Madison, 1991.

[5] Robert S. Boyer, Bernard Elspas, and Karl N. Levitt. Select – a formal system for testing and debugging programs by symbolic execution. In *Proceedings of the International Conference on Reliable Software*, pages 234–245, 1975.

[6] CERT advisory CA-88:01.ftpd.hole.

[7] CERT advisory CA-91:20.rdist.vulnerability.

[8] Juei Chang, Debra J. Richardson, and Sriram Sankar. Structural specification-based testing with ADL. Submitted to ISSTA 1996 as a Regular Paper.

[9] 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*, 15(11):1318–1331, November 1989.

[10] W. F. Clocksin and C. S. Mellish. *Programming in Prolog*. Springer-Verlag, 4th edition, 1994.

[11] Richard A. DeMillo and A. Jefferson Offutt. Constraint-based automatic test data generation. *IEEE Transactions on Software Engineering*, 17(9):900–910, September 1991.

[12] Jeremy Dick and Alain Faivre. *Automating the Generation and Sequencing of Test Cases from Model-Based Specifications*, chapter 4, pages 268–284. First International Symposium of Formal Methods Europe Proceedings. Springer-Verlag, 1993.

[13] Antoni Diller. *Z: An Introduction to Formal Methods*. John Wiley & Sons, 1990.

[14] Michael Edwards and David Bergstein. A look at the current automated capabilities of traceability. In *Proceedings of the IEEE Workshop on Real Time Applications*, pages 204–206, 1993.

[15] Michael D. Ernst. Practical fine-grained static slicing of optimized code. Technical Report MSR-TR-94-14, Microsoft Research, Redmond, WA, July 26, 1994.

[16] George Fink, Michael Helmke, Matt Bishop, and Karl Levitt. An interface language between specifications and testing. Technical Report CSE-95-15, University of California, Davis, 1995.

[17] George Fink, Calvin Ko, Myla Archer, and Karl Levitt. Towards a property-based testing environment with applications to security-critical software. In *Proceedings of the 4th Irvine Software Symposium*, April 1994.

[18] George Fink and Karl Levitt. Property-based testing of privileged programs. In *Tenth Annual Computer Security Applications Conference*, pages 154–163. IEEE Computer Society Press, December 1994.

[19] Charles N. Fischer and Jr. Richard J. Lebanc. *Crafting a Compiler*. The Benjamin/Cummings Publishing Company, 1988.

[20] Keith Brian Gallagher and James R. Lyle. Using program slicing in software maintenance. *IEEE Transactions on Software Engineering*, 17(8):751–761, August 1991.

[21] John B. Goodenough and Susan L. Gerhart. Toward a theory of test data selection. *IEEE Transactions on Software Engineering*, SE-1(12):156–173, June 1975.

[22] R. W. Gray, V. P. Heuring, S. P. Levi, A. M. Sloane, and W. M. Waite. ELI: A complete, flexible compiler construction system. In *Communications of the ACM*, volume 35, pages 121–131, February 1992.

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

[24] John V. Guttag and James J. Horning. *Larch: Langauges and Tools for Formal Specification.* Texts and Monographs in Computer Science. Springer-Verlag, 1993.

[25] Richard Hamlet. Testing programs to detect malicious faults. In *Proceedings of the IFIP Working Conference on Dependable Computing*, pages 162–169, February 1991.

[26] Michael Helmke. A semi-formal approach to the validation of requirements traceability from Z to C. Master's thesis, UC Davis, September 1995.

[27] J.R. Horgan and S. London. Data flow coverage and the C language. In *Proceedings of the Second Symposium on Assessment of Quality Software Development Tools*, pages 2–10, May 1992.

[28] Susan Horwitz. Private correspondence, 1995.

[29] William E. Howden. Methodology for the generation of program test data. *IEEE Transactions on Computers*, C-24(5):554–559, May 1975.

[30] William E. Howden. Validating programs without specifications. In *Proceedings of the ACM SIGSOFT '89 Third Symposium on Software Testing, Analysis, and Verification (TAV3)*, pages 2–9, 1989.

[31] J. C. Huang. Program instrumentation and software testing. *IEEE Computer*, pages 25–32, April 1978.

[32] ISO standard ANSI/ISO 9899-1990.

[33] Jürrgen Börstler and Thorsten Janning. Traceability between requirements and design: A transformational approach. In *16th Annual International Computer Software and Applications Conference*, pages 362–368. IEEE, IEEE, 1992.

[34] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language.* Prentice Hall Software Series. Prentice Hall, Englewood Cliffs, New Jersey, 1978.

[35] Bogdan Korel. Automated software test data generation. *IEEE Transactions on Software Engineering*, 16(8):870–879, August 1990.

[36] Bogdan Korel and Janusz Laski. STAD – a system for testing and debugging: User perspective. In *Second Workshop on Software Testing, Verification, and Analysis*, 1988.

[37] Bogdan Korel and Janusz Laski. Dynamic slicing of computer programs. *Journal of Systems Software*, 13:187–195, 1990.

[38] Carl E. Landwehr, Alan R. Bull, John P. McDermott, and William S. Choi. A taxonomy of computer program security flaws, with examples. Technical Report NRL/FR/5542-93-9591, Naval Research Laboratory, November 1993.

[39] Janusz Laski. Data flow testing in STAD. *Journal of Systems Software*, 12:3–14, 1990.

[40] Janusz Laski. Path expression in data flow program testing. In *Proceedings of the Fourteenth Annual International Computer Software and Applications Conference*, pages 570–576, November 1990.

[41] Panas E. Livadas and Stephen Croll. The C-Ghinsu tool. Technical Report SERC-TR-55-F, University of Florida, December 1991.

[42] Panas E. Livadas and Stephen Croll. Static program slicing. Technical Report SERC-TR-55-F, University of Florida, December 1991.

[43] Panas E. Livadas and Stephen Croll. Program slicing. Technical Report SERC-TR-61-F, University of Florida, October 1992.

[44] R. W. Lo, K. N. Levitt, and R. A. Olsson. Mcf: A malicious code filter. *Computers & Security*, 14(6):541–566, 1995.

[45] Raymond Waiman Lo. *Static Analysis of Programs with Application to Malicious Code Detection.* PhD thesis, University of California, Davis, 1992.

[46] Teresa F. Lunt. Automated audit trail analysis and intrusion detection: A survey. In *Proceedings of the 11th National Computer Security Conference*, October 1988.

[47] Teresa F. Lunt. Ides: An intelligent system for detecting intruders. In *Proceedings of the Symposium: Computer Security, Threat and Countermeasures*, November 1990.

[48] Brian Marick. *Generic Coverage Tool (GCT) User's Guide.* Testing Foundations, 1992.

[49] Simeon C. Ntafos. On required element testing. *IEEE Transactions on Software Engineering*, SE-10(6):795–803, November 1984.

[50] Ronald A. Olsson, Richard H. Crawford, and W. Wilson Ho. A dataflow approach to event-based debugging. *Software - Practice and Experience*, 21(2):209–229, February 1991.

[51] K. J. Ottenstein and L.M. Ottenstein. The program dependence graph in a software development environment. In *Proceedings of the ACM SIGSOFT/SIGPLAN Symposium on Practical Software Development Environments*, pages 177–184, 1984.

[52] T. Reps and G. Rosay. Precise interprocedural chopping. In *Proceedings of the Third ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 41–52, October 1995.

[53] Debra Richardson. TAOS: Testing with analysis and oracle support. In *Proceedings of the 1994 International Symposium on Software Testing and Analysis*, August 1994.

[54] Debra J. Richardson, Stephanie Leif Aha, and T. Owen O'Malley. Specification-based test oracles for reactive systems. In *14th International Conference on Software Engineering*, May 1992.

[55] Debra J. Richardson, Owen O'Malley, and Cindy Tittle. Approaches to specification-based testing. In *Proceedings of the First ACM SIGSOFT '89 Third Symposium on Testing, Analysis, and Verification(TAV3)*, pages 86–96, December 1989.

[56] S. Sankar. *Automatic Runtime Consistency Checking and Debugging of Formally Specified Programs*. PhD thesis, Stanford University, August 1989. Also Stanford University Department of Computer Science Technical Report No. STAN–CS–89–1282, and Computer Systems Laboratory Technical Report No. CSL–TR–89–391.

[57] S. Sankar and R. Hayes. ADL — an interface definition language for specifying and testing software. Technical Report CMU-CS-94-WIDL-1, Carnegie-Mellon University, January 1994.

[58] S. Snapp, J. Brentano, G. Dias, et al. DIDS (distributed intrusion detection system) - motivation, architecture, and an early prototype. In *Proceedings of the 14th National Computer Security Conference*, October 1991.

[59] E. H. Spafford. The internet worm: Crisis and aftermath. *Communications of the ACM*, pages 678–687, June 1989.

[60] Eugene H. Spafford. Common system vulnerabilities. Workshop on Future Directions in Intrusion and Misuses Detection, 1992.

[61] J. M. Spivey. *The Z Notation: A Reference Manual.* Prentice-Hall, London, 2nd edition, 1992.

[62] Andrew S. Tanenbaum. *Operating Systems – Design and Implementation.* Prentice–Hall, 1987.

[63] Frank Tip. A survey of program slicing techniques. *Journal of Programming Languages*, 3(3):121–189, September 1995.

[64] Mark Weiser. Program slicing. *IEEE Transactions on Software Engineering*, SE-10(4):352–375, July 1984.

[65] Elain J. Weyuker and Bingchiang Jeng. Analyzing partition testing strategies. *IEEE Transactions on Software Engineering*, 17(7):703–711, July 1991.

[66] K. Wilken, D. Goodwin, and J. Burger. Compiler and architecture support for low-overhead program profiling. *Software - Practice & Experience*, 1996. To appear.

[67] John B. Wordsworth. *Software Development with Z.* Addison-Wesley, Workingham, England, 1992.