Objective Measures for V&V and Software Reliability

Mark R. Blackburn, Ph.D.
Systems and Software Consortium, Inc.

blackburn@systemsandsoftware.org
(703) 742-7136

Abstract

Consortium members are interested in various aspects of software reliability, including estimating, predicting, measuring, and using approaches that reduce the effort to achieve more predictable levels of reliability in the systems that they produce. This paper discusses an approach proposed with a Consortium member to derive tests from requirement models that can provide an objective basis for predicting software reliability while reducing the cost. The benefits of this approach over traditional approaches to test-case selection support early reliability estimation, while reducing effort to manually identify and select valid test sets, compressing the "execution time" factor in reliability measure calculation. An added benefit is that the test sets will support requirement-to-test traceability, which is often a requirement in high-assurance systems. Finally, this paper argues that domain-based testing is the software domain equivalent to infinite test time in the electrical or mechanical domains.

Contents

Acknowledgments

The author greatly appreciates his colleagues in the member companies and the Consortium that have provided comments on early drafts of this document. In particular, he would like to thank the following people for their thoughtful comments, suggestions, and insights:

- Dale Borja, United Defense
- Rich McCabe, Systems and Software Consortium, Inc.
- Chris Miller, Systems and Software Consortium, Inc.
- Ed Safford, Lockheed Martin
- Mike Siok, Lockheed Martin
- Gerry Ourada, Lockheed Martin

Introduction

Attaining high reliability of software is ever more challenging as system complexity continues to increase. Arguably, the variations in the software development process can be significant from organization to organization as well as person to person. One aspect of software reliability attempts to numerically quantify the uncertainty in the production of software using models to predict reliability and fault content within the software system. Software fault forecasting relies often on measurement data that is derived primarily from testing to predict the reliability of a system. Testing, combined with fault measurement, often is used to calculate software reliability based on different reliability models.

Software reliability engineering (SRE) [Musa 1993] is a popular approach to guide the testing to support predictable reliability and a recommended practice [Schneidewind 2004]. SRE uses operational profiles to characterize how a system will be used. Using an operational profile to guide testing provides a basis to stop testing after the most-used operations have received the most testing for the given test time. Critics of operational testing cite many reasons why systematic testing is superior [Grottke 2001]. Another problem with reliability testing and prediction is that the result is specific to a particular operational profile [Bishop 2002]. Operational profiles do not necessarily characterize all the requirement, design, or implementation paths through a component, and if a path with a fault is never probed, the fault will not be exposed, thereby providing a false estimate of reliability. Other studies indicate that only a small minority of industrial organizations use operational testing [Frankl 1998]. Many organizations use available fault and historical data with reliability models to estimate reliability.

Background

Figure 1 provides a good place to start the discussion related to test data and reliability estimation. Conceptually, if an organization captures the failures per unit of time throughout the development and test phases, and at the point where the number of failures during that period of time goes below some threshold, then it may be assumed that the reliability of that particular

component is acceptable, and it can be released. From some perspectives, software can have faults, and if no failures manifest during test and usage, then the software is considered reliable. This could be a bad assumption for several reasons:

- For many complex systems today, the environment and operational usage of a software system in the field evolves raising the likelihood that unexposed faults may result in failures in the future.
- The number of failures is not necessarily directly related to the number of faults—for reliability, it is important to identify and remove all the faults to increase the reliability.
- To find all the faults in a software component, test cases must be selected to probe each path through the component's software. Usage-based tests often do not exercise all paths.
- Even when the path is probed by a test case, a fault may not be exposed unless certain critical values are used to expose the fault; therefore, selecting appropriate test data is important.

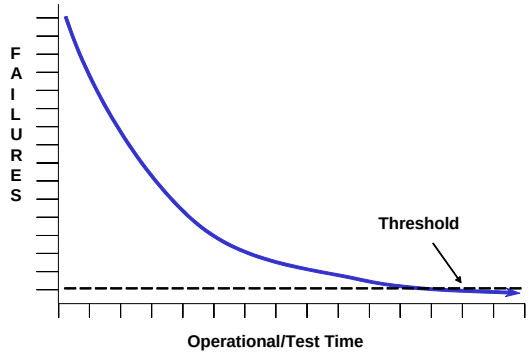


Figure 1. Conceptual Approach

One of the key difficulties in estimating the reliability is directly related to the "goodness" (i.e., fault-finding capability) of the test cases. Software reliability is traditionally based on some form of statistical testing process. It is common to characterize operational profiles and select test sets manually using some type of statistical process. The cost of attaining a statistically sound reliability measure is traditionally high. It is costly to develop enough tests and typically difficult to run all tests and analyze all results when it is done manually. As shown in Figure 2, the cost to achieve high levels of reliability can go up exponentially as the threshold gets closer to 100% reliability, usually represented by 1.0.

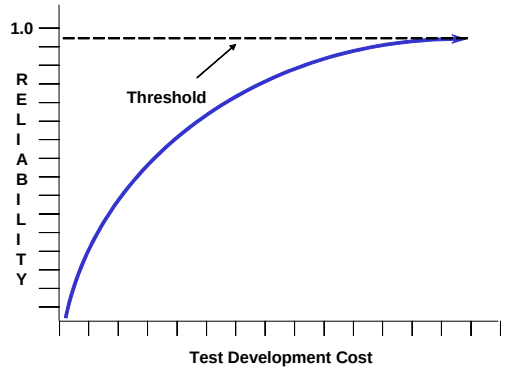


Figure 2. Reliability Cost

Therefore, it is essential to choose a valid subset of tests to support reliability arguments. Unfortunately, the adequacy of manually selected tests cases can vary significantly. If the test sets are not adequate, then the reliability estimate may be unrealistically high because the tests were not effective in uncovering faults.

Approach

This paper discusses an objective approach, based on the use of model-based test tools, to generate test data using a technique based on domain testing to select the most effective test values to uncover faults. Model-based test generation reduces or eliminates manual identification and selection of test sets to significantly reduce the cost and effort. It reduces or eliminates effort dedicated to the traditional approach of identifying and characterizing operational profiles, which can arguably be subsumed by models of requirements, or implementation-derived requirements. Test generation and automated test execution compresses the “execution time” factor in reliability measure calculation, which can start and proceed concurrently throughout development.

Modeling and test generation minimize the process variation between individuals and organizations and provides an additional advantage of finding requirement or design faults. Software faults are all design faults [Storey 1996]. Software faults result from human error in interpreting or creating requirement or design specifications or incorrectly implementing the specification within a component. Tests generated from a model of the requirements or design provide a means for exposing a fault in the implementation, but the use of models and the model checking that is performed as part of the automatic test generation are effective in exposing requirement or design defects [Blackburn 2004b]. The recommended uses of the model-based testing proceed in parallel with the development [Blackburn 2003]. Finally, the potential for representing usage information (e.g., use cases, operational profiles) into models that can be used to support automated model-based testing has been demonstrated [McCabe 2002].

Scope

This paper discusses an approach to generate test cases that provide a more objective basis for estimating the reliability of a software component or system. The approach is based on using test suites derived by domain and subdomain-based test generation techniques that are associated with the requirements or implementation-derived requirements of a software-intensive system. The paper discusses the following:

- Why the domain-based tests are the minimal set of tests necessary for full coverage of the paths through a software implementation
- Why this coverage provides an objectively arguable minimal set of tests
- The test results effectiveness from an existing Consortium system where the model-based results were compared to actual usage-based test results
- The results of an experiment that has shown that domain-based tests with full code coverage are significantly smaller in size than statistical test sets
- Why full-coverage, domain-based testing is the software domain equivalent to infinite test time in the electrical or mechanical domains.

The model-based tools discussed in this paper are part of the integrated environment generically referred to as the Test Automation Framework (TAF) that integrates government and commercially available model development and test generation tools. One of the latest additions to TAF integrates the DOORS requirement management tool with the T-VEC Tabular Modeler (TTM) that supports the Software Cost Reduction

(SCR) method [Heitmeyer 1996] for requirement modeling. DOORS integrates also with Simulink, which supports design-based models, and TAF integrates requirement models with design models to provide full traceability from the requirements source to the generated tests, as reflected in Figure 3.

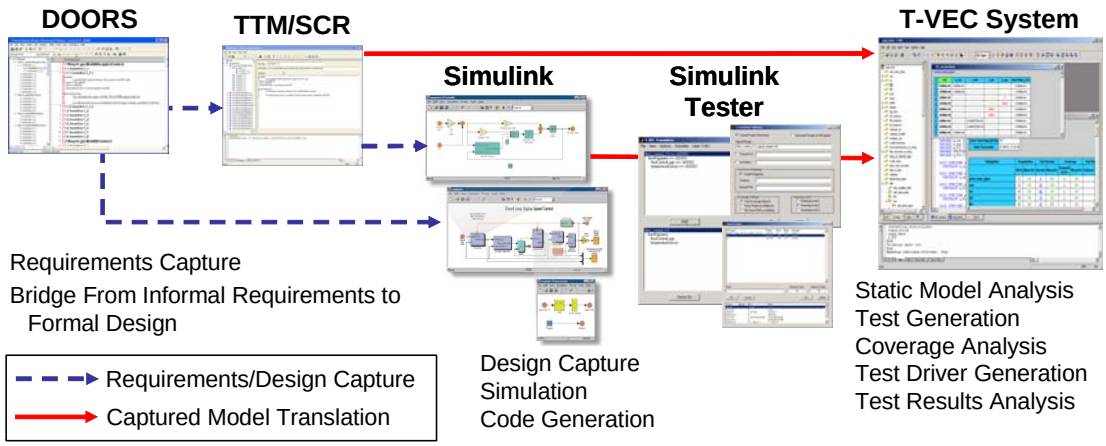


Figure 3. TAF Integrated Environment

Audience and Benefits

The audience for this deliverable consists of member companies that are interested in an approach for objectively predicting software reliability through a systematic approach to software testing, based on models. The paper provides information on how to better leverage testing results to support a more systematic approach to predictable reliability. Given the cost and time constraints of today's competitive market, the effort required to support requirement-driven, test-set selection supports early estimation and better prediction of the time required to achieve predictable reliability. Organizations considering the use of model-based testing are provided with additional arguments supporting reliability estimation, in addition to information on how the test sets will support requirement-to-test traceability, which is often a requirement in high-assurance systems, with the benefit of reduced cost of maintainability.

A key benefit is that the proposed approach supports nontime-based reliability estimation. Simply stated, once tests are created and executed for every path of the software component associated with a modeled requirement, the reliability measure is essentially equivalent to infinite time models related to hardware and mechanical devices.

Key topics covered by this report include:

- Context and definitions associated with software reliability (See Context and Definitions)
- Key issues related to test data effectiveness and two examples that illustrate the impact on reliability (See Test Data Effectiveness)
- Technical details describing fault-finding effectiveness of a model-based test generation approach that subsumes domain-based test selection (See Technical Details)

- Experiment details that could be used to test other potential approaches (See Experimental Details)
- Summary and some other potential benefits (See Summary and Conclusions)

Context and Definitions

Definitions

Reliability is an attribute of dependability as shown in Figure 4. Dependability is a collective term subsuming the notions of reliability, availability, safety, confidentiality, integrity, maintainability, and security [Laprie 1984]. The means to achieve it are based on fault prevention, fault tolerance, fault removal, and fault forecasting to avoid threats. A system is completely dependable if it never **fails**. A **failure** results from a fault. A **fault** is caused by a human that makes an error in the specification (i.e., requirement or design) or implementation of system artifacts. There are many categories of faults—for example, a design fault occurs if a requirement is represented incorrectly in the system. (For more information on other facets of dependability see *Guidance for Achieving Mission Assurance in Software-Intensive Systems* [Blackburn 2004a]).

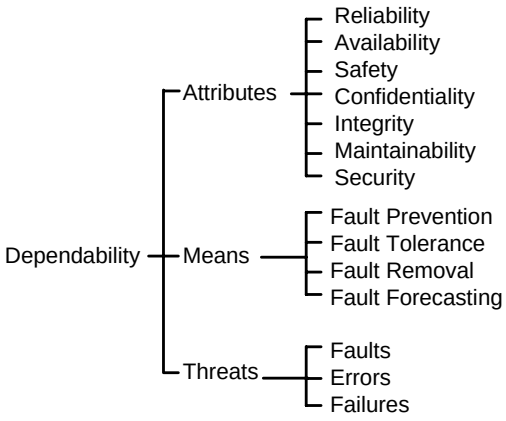


Figure 4. The Dependability Tree [AMSD 2003]

Reliability, is the probability that software will not cause the failure of a system for a specified time under specified conditions. The probability is a function of the inputs to and use of the system, as well as a function of the existence of faults in the software. The inputs to the system determine whether existing faults, if any, are encountered [IEEE 1988]. Highly reliable systems are used in situations in which repair cannot take place (e.g., spacecraft) or in which the computer is performing a critical function for which even the small amount of time lost because of repairs cannot be tolerated (e.g., flight-control computers).

Availability is the intuitive sense of reliability. A system is available if it is able to perform its intended function at the moment the function is required. Formally, the availability of a system as a function of time is the probability that the system is operational at that instant of time. Availability is frequently used as a figure of merit in systems for which service can be delayed or denied for short periods without serious consequences.

Safety is the absence of catastrophic consequences on the user(s) and the environment.

Confidentiality is the absence of unauthorized disclosure of information.

Integrity is the absence of improper system alterations.

Maintainability is the ability to undergo repairs and evolutions.

Security is the absence of unauthorized access to, or handling of, system state and relates to availability, confidentiality, and integrity.

Organizations must use an engineering approach to satisfy these dependability properties by avoiding faults in the requirements, design, and implementation of the target system. Figure 5 shows a taxonomy to relate the engineering approaches for fault avoidance [Lyu 1995]. The approaches include fault prevention, fault removal, fault tolerance, and fault forecasting. The types of faults that are relevant to software primarily include design, interaction, and malicious logic faults.

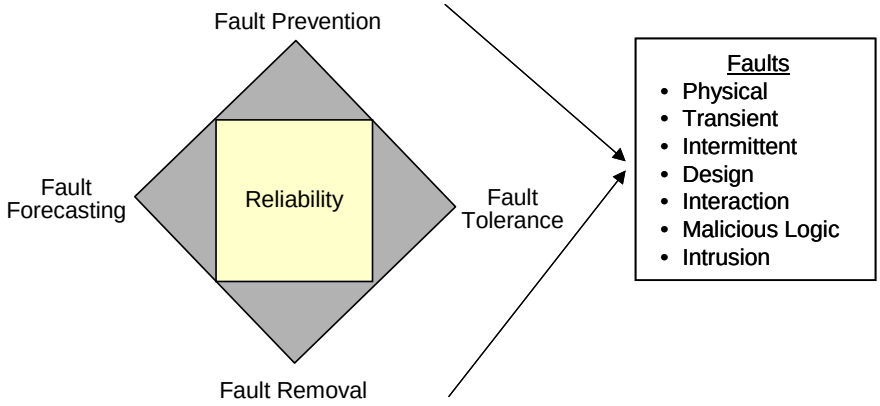


Figure 5. Fault Taxonomy

Software Reliability Models

Reliability analysis attempts to estimate the probability of losing a function as a result of component failures. Reliability analysis typically uses physical component models characterized by failure-rate statistics, where the goal of the analysis is to calculate system-level failure rates for selected functionalities and to determine which component faults contribute to the loss of the selected functionalities. A number of software reliability models have been proposed for assessing the reliability of a software system, some based on the time domain and other based on the data domain [Gokhale 1996].

Time-domain models represent the underlying failure process of the software under consideration and use the observed failure history as a guideline in order to estimate the residual number of faults in the software and the test time required to detect them. The

time-domain models are classified into homogeneous Markov,[1] nonhomogeneous Markov, and semi-Markov models.

There is debate on the validity of time-related software reliability models because of problems when the software's environment changes. This particularly affects commercial off-the-shelf (COTS) and Software of Unknown Pedigree (SOUP) because a failure may occur for a first time in a new environment if the software does not exactly match the intended functionality [Advantage1 2002]—for example, the Ariane 5 failure. Such failures make any time factor irrelevant. Thus, time-related reliability models for COTS or SOUP are difficult to justify for high-assurance (e.g., safety-critical) systems, and additional evidence or approaches are required [Caseley 2003].

Data-domain models are based on the philosophy that if the set of all input combinations to a computer program are identified, an estimate of the program's reliability can be obtained by exercising all the input combinations and observing the outcomes. In practice, it is not feasible to identify the set of all input combinations, and this approach is reduced to a method of selecting sample data sets representative of the expected operational usage for the purpose of estimating the residual number of faults in the software product under consideration. Data-domain models can be further classified as:

- **Fault-seeding models.** The software product, which has an unknown number of indigenous faults, is seeded with a known number of faults and subjected to rigorous testing. An estimate of the actual number of indigenous faults is then obtained by determining the ratio of discovered seeded faults to discovered actual faults. (More information on this approach is provided in the section on Experimental Details.)
- **Input-domain models.** In the case of input-domain models, the reliability of the software is measured by exercising the software with a set of randomly chosen inputs. The ratio of the number of inputs that result in successful execution to the total number of inputs gives an estimate of the reliability of the software product.

Test Data Effectiveness

This section provides two examples to illustrate the importance of test data selection as it relates to reliability estimates. For data-domain and time-domain models, the adequacy of the tests is critical to the reliability estimate. Although the typical approach referenced in the context of software reliability models is to generate test cases by selecting operations randomly (i.e., statistically) according to the operational profile and input states randomly with their domain [Musa 1993], this paper proposes to leverage automated tools to provide domain-based test data that is more effective than statistically selected data.

---

[1] Andrey Markov was the Russian mathematician who helped to develop the theory of stochastic processes, especially those called Markov chains, which are based on the study of the probability of mutually dependent events.
[http://www.britannica.com/search?query=Markov+&submit=Find&source=MWTEXT]

Example 1

Table 1 shows the result of an experiment that compares the effectiveness of domain-based tests generated by TAF/T-VEC versus statistically based (random) tests. (See Experimental Details for the details of the experiment.) There were two different versions of the base program, version 1 with 39 lines of code and 17 decisions that was seeded with 112 faults, and version 2 with 42 lines of code and 19 decisions was seeded with 117 faults. Based on the effectiveness score that is derived from computing the number of seeded faults that were exposed by the test cases with respect to the total number of seeded faults, the two model-based test vector sets had scores of 98.2 % and 99.1 % as compared to the respective scores of 46.4 % and 62.4 % for the statistically-generated test sets. The model-based test sets provided 100 % decision coverage, and the statistically generated test sets resulted in 89.4 % and 90.6 % coverage, even though the effectiveness scores were low. Even with nearly twice as many statistically based test cases and relatively high decision coverage (e.g., path coverage), the results suggest that the approach that uses domain-based test data is critical to uncovering faults.

Table 1. Experiment Data

Experiment Parameters	Version 1		Version 2	
Lines of Code	39		42	
Decisions	17		19	
Test Selection Approach	Domain	Statistical	Domain	Statistical
Test Cases	97	176	105	208
Seeded faults	112	112	117	117
Exposed faults	110	52	116	73
Complete Decision Coverage	100.0%	89.4%	100.0%	90.6%
Effectiveness Score	98.2%	46.4%	99.1%	62.4%

Example 2

Consider the example represented in Figure 6, which is distilled from working with a Consortium member. During the integration testing of two software-system components, test cases were selected, and data points from the input spaces were used to test the functionality. The test-case selection process was based on the operational usage; however, critical faults were not identified. The requirements for the functionality were later modeled using TAF, and tests were generated based on the use of domain and subdomain test selection principles, where test points are selected at the boundary of the input space. These tests resulted in 318 test cases, with 152 failures. The test cases generated from the model exposed faults. Analysis helped determined that the values generated by T-VEC, the generation component of TAF, were located at the boundaries or subdomain boundaries of the input space, and these particular values exposed the faults. The manually selected values were selected based on usage data, from data points within the subdomains and did not identify any faults.

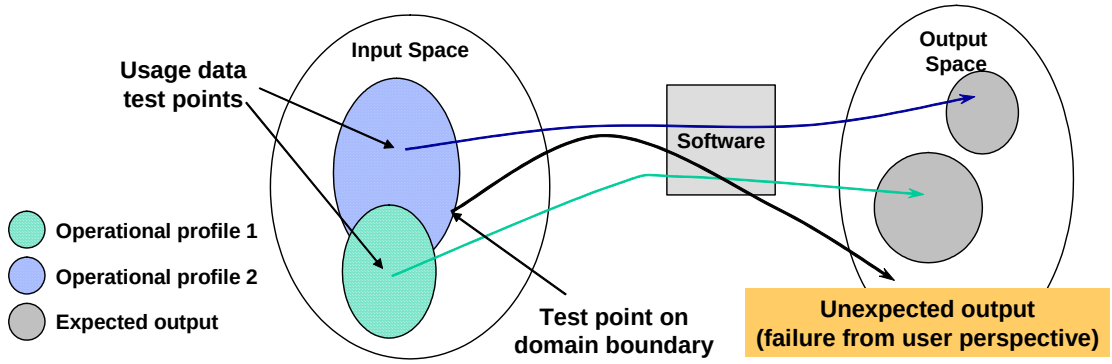


Figure 6. Conceptual Representation of Limitation of Statistical Testing

Testing Strategies

Testing strategies are typically characterized as **structural**, that is, related directly to an implementation, and **functional**, that is, based on a specification [Howden 1980]. Specification-based testing, or black-box testing, relies on properties of the software that are captured in the functional specifications (or requirements model, including interfaces, possibly design and other behavioral information). The traditional functional testing approach is to partition the input domain into equivalence classes and select test data from each class [Richardson 1992].

The following three types of errors can be made in an implementation, resulting in a fault [Howden 1976]:

- A ***computation error*** occurs when the correct path through the program is taken, but the output is incorrect because of faults in the computation along the path.
- A ***domain error*** occurs when an incorrect output is generated because the wrong path was executed through a program.
- A ***missing-path error*** occurs when the implementation does not fully implement the requirements or design. This type of error must be identified by inspection rather than testing.

[White 1980] proposed ***domain testing theory*** as a strategy for selecting test points to reveal domain errors. It is based on the premise that if there is no coincidental correctness, then test cases that localize the boundaries of domains with arbitrarily high precision are sufficient to test all the points in the domain. Domain testing theory is based on the intuitive idea that faults in the implementation are more likely to be found by test points chosen near appropriately defined program input and output domain boundaries [Tsai 1990].

[Beizer 1995] describes "domain testing" as an approach to select test inputs based on the domains (ranges) of the system inputs at or around the maximum and minimum values of these ranges. Furthermore, he describes "formal specification based testing," which starts with the system input domains, then partitions these domains into subdomains based on the logic within the specification (model). Using this technique,

picking test input values from the subdomains verifies the logic of the application. This is also referred to as partitioning. Each partition is an equivalence class that must be tested.

As noted the Introduction, SRE is one of the noted approaches of software reliability engineering, based on identifying operational profiles (sometimes related to use-case testing). An operational profile is a usage model that characterizes possible scenarios of software use at a given level of abstraction. A random test case is a traversal of the usage model based on state transitions that are randomly selected from a usage probability distribution. The notion of a random test value is that a statistically independent set of values is selected from an input domain. Random test sets are a form of statistical testing that can be based on various non stochastic (i.e., probabilistic) and stochastic models. **Statistical testing** is based on the premise that when a population is too large for exhaustive study, as is the case for all possible uses of a software system, a statistically correct sample must be drawn from the input population [Poore 1998].

Limits of Usage-Based Testing

The concern with usage-based tests, like SRE, is that they seldom cover all the paths through the component. A path results from constraints in the requirements or design decisions associated with implementation-derived requirements that are implemented as decisions (i.e., control-flow paths) in the implementation. The constraints result in input subdomains associated with a path through the implementation. The complexity of systems today typically results in many layers of implementation-derived requirements. Each component has one or more paths, and each path must be tested to provide assurance that there are no faults. If a path is not probed by at least one test, a fault cannot be detected; however, full test coverage does not guarantee detection of all faults because the proper test inputs often are required in order to expose a fault as illustrated by the results shown in Example 1 and Example 2. The second problem is that the values selected by usage tests often are nominal values, not at or near domain or subdomain boundaries, where failures are more likely to be exposed.

Technical Details

This section uses an example to explain technical details related to test data selection for domains and subdomains. Although the test selection process can be performed manually, the number of cases and the complexity of system requirements make the selection of test values at the domain or subdomain boundaries a challenging manual task. TAF's test generation performs domain-based test selection from a requirements model of the software component under test or a hierarchical model to address a multileveled system component. The test generation produces test points for all constraints in a model and can provide the equivalent of modified condition-decision-level [MCDC] test coverage of the modeled specification. For MCDC coverage, every point of entry and exit in the program has been invoked at least once; every condition in a decision in the program has taken on all possible outcomes at least once; and each condition has been shown to affect that decision outcome independently. A condition is shown to affect a decision's outcome independently by varying just that decision while holding fixed all other possible conditions. See [Hayhurst 2001] for a more in-depth summary of structural coverage.

Understanding Domains and Subdomains

The following example explains the concepts of domain and subdomain as they support an objective approach to systematic test selection. The following Vertical Tracker example is simplified from a requirement of the Traffic and Collision Avoidance System (TCAS). A Vertical Tracker component would track another aircraft relative to one's current altitude and must maintain the tracking state. Figure 7 provides some additional details about the Vertical Tracker. The altitude of an aircraft is defined by a standard known as Gilliam Altitude, and it can range from –1,000 feet to 127,000 feet. The Own Aircraft is located in the center of the model. Another aircraft is considered to be "in_altitude_window" if it is within 2,700 feet above or below Own Aircraft. It is within the altitude rate limit (i.e., "rate_limit") if it satisfies the constraint that is represented by the shaded area. Other requirements are related to the Vertical Tracking state as defined by the following:

- The Vertical Tracker for Own Aircraft shall be in TRACKING state if Own Aircraft is at or above 10,000 feet in altitude, but no other aircraft is in the altitude window.
- The Vertical Tracker for Own Aircraft shall be in ADVISORY state if Own Aircraft is at or above 10,000 feet in altitude, and another aircraft is in the altitude window.
- The Vertical Tracker for Own Aircraft shall be in NOT_TRACKING state if Own Aircraft is less than 10,000 feet in altitude.

As shown in Figure 7, Aircraft 1 is in the altitude window but does not meet the altitude rate limit; Aircraft 2 meets both constraints; and Aircraft 3 does not meet any of the constraints.

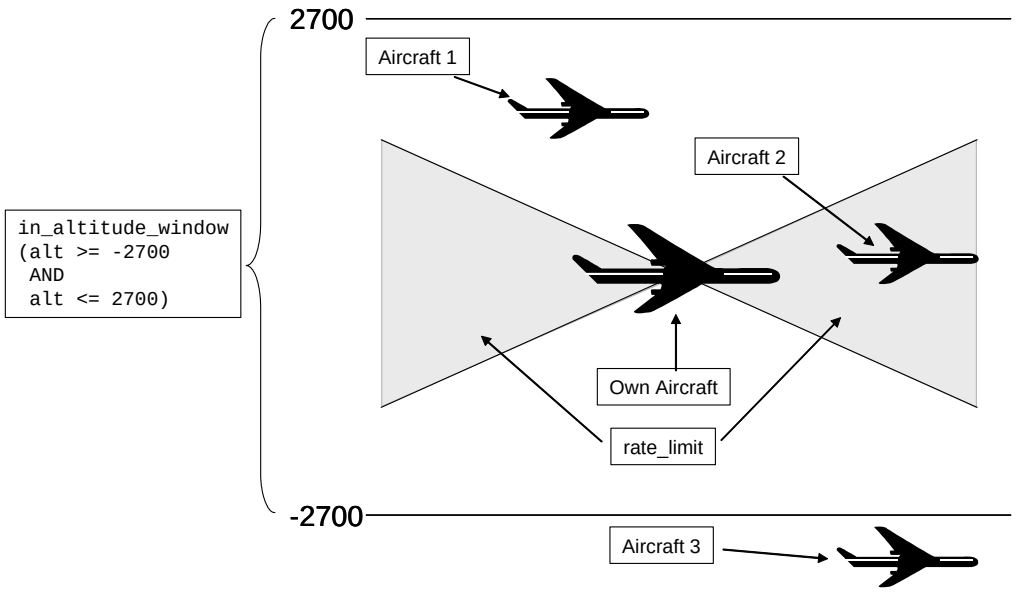


Figure 7. Visual Representation of Vertical Tracker Constraints

The domain is the set of values that go into a function; the domain of the input space for aircraft altitude can range from –1,000 to 127,000 feet. For two aircraft, the input space is defined by the cross-product of Own Aircraft's altitude and the nearest aircraft's

altitude (i.e., “Nearest_altitude”), and both have the same input domain as reflected in Figure 8. For this example, Aircraft 2 is the Nearest_altitude, as shown in Figure 7.

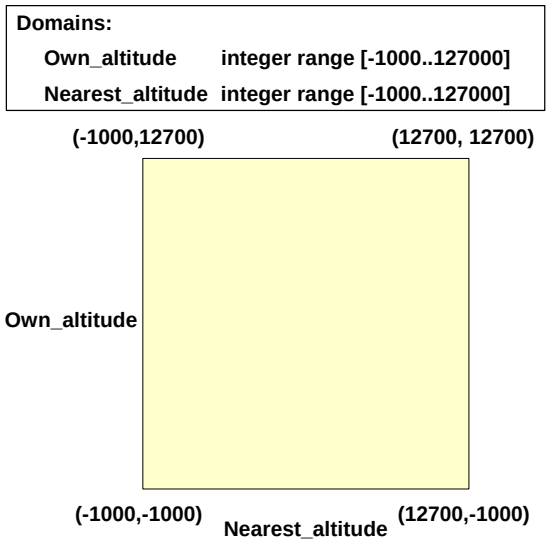


Figure 8. Input Domains

Another requirement for the Vertical Tracker is that if the Own Aircraft is above 10,000 feet, then it is in tracking mode. This requirement imposes a constraint on the input space that partitions the domain into two subdomains as reflected by Figure 9, labeled TRACKING MODE and NOT TRACKING MODE. For testing purposes, it is important to have at least one test point for each subdomain of a component’s input space. Ideally, the test points should be selected near the boundaries of the domain. In this case, some of the critical points for the boundary defined by the constraint occur where the Own_altitude is less than 10,000 (e.g., –1,000, 9,999) and greater than or equal to 10,000 (e.g., 10,000, 10,001, 12,700).

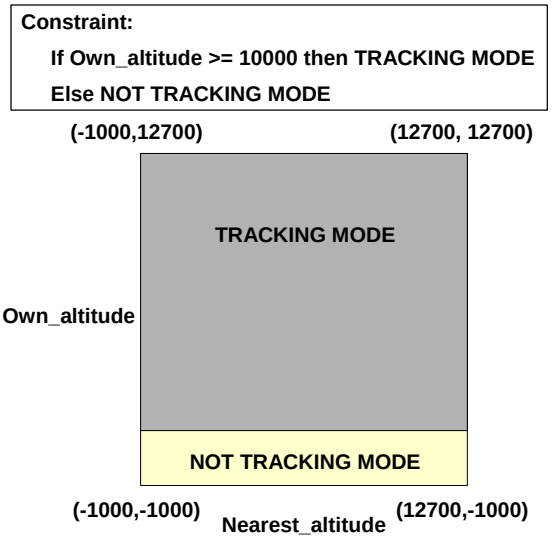


Figure 9. Partition of Domain Into Subdomains

When the Own Aircraft is in tracking mode (i.e., Own_altitude >= 10,000) and another aircraft is within 2,700 feet above or below, then the aircraft goes into an advisory mode.

This creates another subdomain, labeled ADVISORY, with different boundaries as reflected in Figure 10. Figure 11 shows test points overlaid on an enlarged image of the ADVISORY subdomain for test cases 11, 13, and 18 that were generated by TAF. Notice that the values selected are at or near the boundaries of the subdomain. These are the test values that are likely to uncover faults.

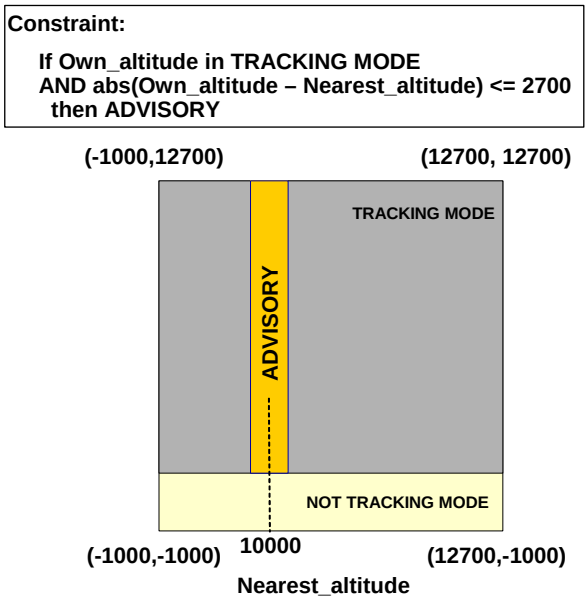
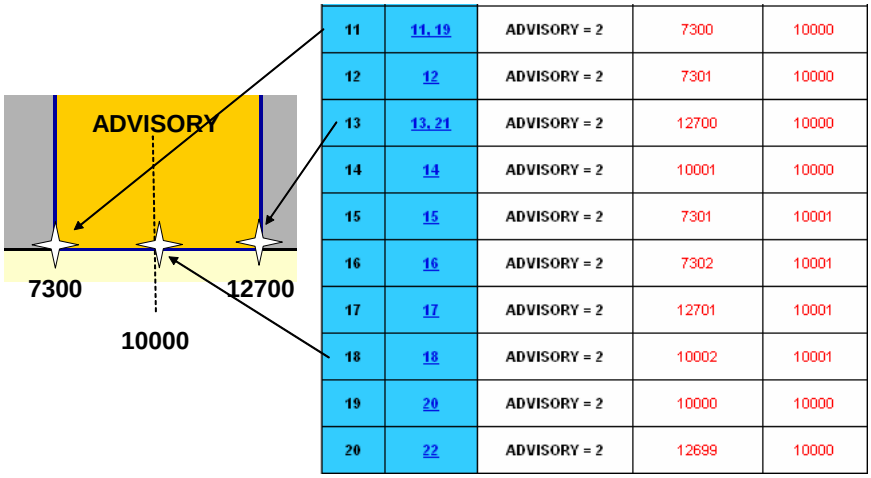


Figure 10. Constraints for Advisory Mode



11	11, 19	ADVISORY = 2	7300	10000
12	12	ADVISORY = 2	7301	10000
13	13, 21	ADVISORY = 2	12700	10000
14	14	ADVISORY = 2	10001	10000
15	15	ADVISORY = 2	7301	10001
16	16	ADVISORY = 2	7302	10001
17	17	ADVISORY = 2	12701	10001
18	18	ADVISORY = 2	10002	10001
19	20	ADVISORY = 2	10000	10000
20	22	ADVISORY = 2	12699	10000

Figure 11. Sample Test Points

Domain Error Coverage

Subdomains are introduced when a requirement or design constraint is specified. These constraints result in decisions that must be implemented, and those decisions manifest in different code paths. The constraints introduce new boundaries as shown in Figure 10, which are associated with the logically AND'ed set of conditions for that constraint, for example:

The Vertical Tracker is in ADVISORY mode when Own Altitude >= 10,000 AND

abs(Own Altitude – Nearest Altitude) <= 2,700

In T-VEC, this is call a domain convergence path (DCP), and the test generator selects input test values for the borders of the subdomain. A **border** is defined by evaluating the predicates (i.e., individual constraints) of a DCP for a set of input values. Test points for numeric objects are selected for both upper and lower domain boundary values as reflected in Figure 11. This results in test points for subdomain borders based on all low-bound values and high-bound input values that satisfy the DCP predicates.

Domain errors are associated with the requirements and occur if the constraints are not satisfiable. If domain errors do not exist in the modeled requirement, the selection of extreme values provides test points to detect computation errors (e.g., overflow, underflow, or incorrect calculations). Intuitively, this domain testing mechanism provides confidence that every path of the implementation is correct with respect to every DCP and function of a component's requirement or design model.

Table 2 shows a few of the test points that should be considered for the Vertical Tracker. These values are near the domain or subdomain boundaries.

Table 2. Test Points for Vertical Tracker State

#	State	nearest_altitude	own_altitude
1	NOT_TRACKING	127000	-1000
2	NOT_TRACKING	-1000	9999
3	TRACKING	-1000	10000
4	TRACKING	12501	10000
5	TRACKING	124499	127000
6	TRACKING	127000	124499
7	ADVISORY	7500	10000
8	ADVISORY	10001	10000
9	ADVISORY	127000	127000
10	ADVISORY	127000	126999

Computation Error Coverage

Some inputs to functions are not constrained by the DCP predicates. For each test point derived from DCP predicates, there are additional test points derived for unconstrained inputs not referenced in the DCP. Test values are selected based on the domain boundary value combinations (e.g., low bound and high bound for numeric objects and sets for enumerated variables). By selecting the extreme value combinations, there is a possibility to detect computation errors in the output calculation. This test selection strategy is used to detect computation errors or show that unconstrained inputs do not affect the output for a program path.

Path Combinations in Hierarchical Systems

Test effectiveness in hierarchical systems is more challenging because paths are threaded together through hierarchically related components. A path refers to one logically AND'ed set of constraints as discussed for subdomains, but with hierarchical paths, it includes logically AND'ing conditions from hierarchically related components. Usage data is not likely to cover all paths related to the implementation or the requirements. There are two key issues related to selecting effective test data for hierarchically related components:

- Selecting test data to cover all combinations of paths requires a combinatorial larger number of test cases.
- Selecting test values to traverse all paths is numerically challenging, requiring that constraints associated with the paths be analyzed to select the appropriate test data.

To generate tests for every path throughout a hierarchy of software components can be more effort-intensive and costly, but the cost and effort to minimally cover each path through all components can be minimized if testing is performed on a layer-by-layer basis. Consider the example shown in Figure 12:

Component C calls C.1, and C.1 calls C.1.1 and C.1.2. The exclusive paths in each component are:

- C = 2 paths
- C.1 = 4 paths
- C.1.1 = 4 paths
- C.1.2 = 3 paths

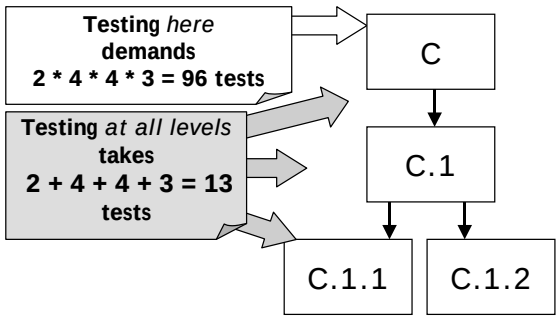


Figure 12. Example Illustrating Paths and Test Coverage

To cover all the paths from the top-level component (C) would require 96 total tests. The required number of tests is 13 if a level-by-level approach is used to test each path in each component. The effort becomes exponential if a tester attempts to test all combinations of paths from the higher level, for example through a graphical user interface (GUI) represented by component C. When testing on a level-by-level basis, tests must be constructed to cover only the conditions associated with the paths of that component, which also can demonstrate that the integration within that level operates properly. Usage-based testing seldom occurs on a level-by-level basis, and it is unlikely that 96 tests will be performed. Therefore, the reliability estimate should take this into consideration.

TAF supports hierarchical relationships. In generating test vectors for a hierarchy of models, as represented in Figure 13, the test generator selects test cases for the DCP paths of the high-level components (e.g., Grandparent) without regenerating all the test vectors for each referenced lower-level subsystem. The test vector generator bases the test selection on the DCPs for the upper-level subsystem (Grandparent), not the combination of DCPs for the parent and children subsystems. This mechanism precludes the combinatorial explosion associated with tests generated from the combination of constraints in a hierarchy of subsystems as represented in Figure 12.

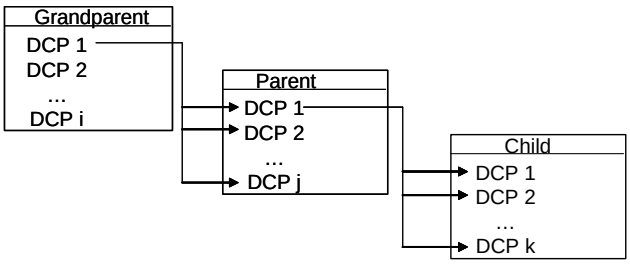


Figure 13. Hierarchical Subsystem Relationships

This level-by-level process provides an efficient means for performing unit, software-integration, and system-level testing. For additional information on this recommended process, see Section 4 of *Guidance for Achieving Mission Assurance in Software-Intensive Systems* [Blackburn 2004a]. This report recommends a recursive process for specifying functional requirements, design, and architectures combined with a continuous and layered approach to verification so that requirement and design artifacts are verified and defects are contained to their phase of creation, while systematically applying verification processes at each layer of the system.

Experimental Details

This section describes an experiment to test the hypothesis that the model-based testing approach using domain-based test selection is more effective at finding faults than statistically based test-set selection. The experiment compares the TAF/T-VEC model-based test generator with a statistically based test generator and uses a fault-seeding technique, referred to as mutation testing, with path-coverage analysis techniques to assess the adequacy of the generated test sets. The results, shown in Table 1, suggest that the evaluated model-based test generator is better than 98 % effective at finding program faults and is as much as 51 % more effective than a random number test-set generator with half the number of test cases. The implications of the results are that the model-based test generation approach provides a more systematic approach to test selection that can support better reliability estimation in a predictable set of time that is directly related to the modeled requirements. Another key finding is that the statistically selected test cases had a relatively high degree of decision coverage, but that did not make them effective in finding the seeded bug. This supports the argument that it is important to select critical values to uncover faults.

Mutation testing is a fault-based testing technique that has been effective in assessing the adequacy of a test set for a program [Hamlet 1977; DeMillo 1978]. For any program, **mutations** of a base program (referred to as mutants) are generated through the use of mutation operators. A **mutation operator** describes a set of syntactic changes based on

program language constructs. Each mutant contains one fault. The adequacy of a test set can be measured by its ability to detect the mutants derived from the base program. A **mutation score** for a test set is the percentage of nonequivalent mutants that are killed (i.e., detected) by a test set.

Mutation testing has been shown to be effective; however, because the number of mutants for real-world programs can grow large, it is typically expensive to use. This experiment used selective mutants [Offutt 1994]. **Selective mutants** use a subset of the standard mutation operators and appear to be effective in generating minimally sized, adequate test sets for finding faults in programs.

The **coverage criterion** applied to the subjects was **decision coverage**: every point of entry and exit in the program has been invoked at least once, and every decision in the program has taken on all possible outcomes at least once [RTCA 1992]. The base programs were instrumented so that every decision and every statement guarded by a decision had a statement to record that a path through the program had been executed. The instrumented statements also were included automatically when the mutants were generated but were not modified by the mutant generator. The test set was run through a mutant, and those instrumented locations that were not executed were recorded programmatically.

Figure 14 shows the relationships between the elements of the design. The design uses a **base program** that is a correct implementation for the requirements. The specification was modeled using TAF. Test vectors were generated from the model using T-VEC. The base program executes correctly with respect to the requirements. For this experiment, **correct** means that the:

- Actual output values must equal the expected output values for all test inputs.
- Test sets must satisfy complete program coverage based on a coverage criterion.

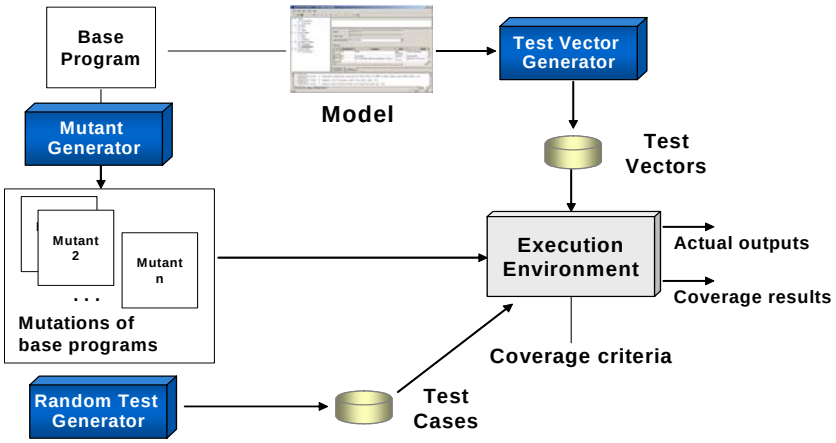


Figure 14. Experimental Design

The base program was mutated based on the selective mutants. The selective mutant operators summarized in Table 3 were applied to those constructs used in two different

base program versions. The mutations resulted in 112 mutants for version 1 and 117 mutants for version 2. The test sets, both generated and random, were executed by each mutant, and the actual output and coverage results were recorded.

Table 3. Select Mutant Operators

Function	Mutations Applied to Base Program Subjects
Absolute value insertion	Each arithmetic expression to take on the value 0, a positive value, and a negative value
Arithmetic operator replacement	Replaces every syntactically legal operator ('+','-','*','/')
Logical connector replacement	Replaces each logical connector (AND and OR)
Relational operator replacement	Replaces relational operators with other syntactically legal relational operators ('<=', '>=', '<', '>', '!=')
Unary operator insertion	Inserts unary operators in front of expressions (replaces a '-' with a '+', or inserts a '-' in front of expressions)

Summary and Conclusions

This paper discusses a systematic approach to test-case definition that should provide a more objective basis for estimating the reliability of a software component. Recommended practices to reliability estimation and prediction often are based on test selection guided by operational profiles, but operational profiles seldom cover all the paths through the component. A path results from requirement constraints or design decisions that are implemented as decisions in the implementation. The requirement constraints manifest in paths through the implementation. The complexity of systems today typically results in many layers of implementation-derived requirements. Each component has one or more paths. To maximize the confidence in the reliability estimates, each path must have one associated test to provide assurance that there are no faults in the decision or computations of that path. If a path is not probed by at least one test, a fault cannot be detected. However, experiment data presented in this paper shows that full test coverage over all paths does not guarantee detection of all faults because the proper test inputs often are critical in order to expose a fault. Recognizing that the assumption for software reliability models is to generate test cases by selecting operations randomly (i.e., statistically) according to the operational profile and input states randomly with their domain, this paper recommends the use of test data values generated from requirement models that localize the test values at or near the domain and subdomains boundaries, where faults are more likely to be exposed.

Benefits

The benefits of this approach over traditional approaches to test-case selection to support reliability estimation include the following:

- Reduces or eliminates the activity of manually identifying and selecting minimally valid test sets
- Compresses the “execution time” factor in reliability measure calculation
- Reduces or eliminates the effort dedicated to identifying and characterizing operational profiles that can be subsumed by models

- Eliminates manual determination of operational profiles and associated test suites by subsuming them with domain- and subdomain-based tests
- Supports early estimation and better prediction of the time and effort required to achieve predictable reliability, with the added benefit that the test sets will support requirement-to-test traceability, which is often a requirement in high-assurance systems

Since 1996, the Systems and Software Consortium, Inc. (SSCI) has worked with Consortium members and continues to see increased adoption of model-based testing tools. Model-based testing has many benefits, including better quality requirements, better tests, and faster test design. Modeling and test generation minimize the process variation between individuals and organizations and provide additional capabilities to address software reliability with additional tools and side benefits. Tests generated from a model of the requirements or design provide a means for exposing a fault in the implementation, but the use of models and the model checking that is performed as part of the automatic test generation are effective in exposing requirement or design defects.

With the recent integration of requirement management tools with SSCI's requirement and design-based modeling and test generation tools, the tools support full requirement-to-test traceability. Some of the key benefits derived through the traceability process help to foster organizational adoption of the model-based testing tools that provide the benefits of the systematic test generation that might be more effective in helping organizations quantify their reliability earlier in a nontime-based approach to reliability estimation as they directly relate their reliability estimates in the context of requirement coverage and completeness.

References

[AMSD 2003] AMSD Partners, University of Newcastle. Tom Anderson, ARCS; Erwin Schoitsh, CNUCE; Luca Simoncini, LAAS; Jean Claude Laprie; JRC; Marc Wilikens, Adelard; and George Cleland. "Accompanying Measure in System Dependability." FP5.8 KAII Road mapping project, June 2002 - May 2003.

[Advantage1 2002] Advantage1. *The Assessment of Software Components for Safety Applications – Management Guidance*, 21980/05/Rep-02, Issue 1.Available through the MoD Acquisition Management System. Advantage1, March 2002.

[Beizer 1995] Beizer, B., Black-Box Testing: Techniques for Functional Testing of Software and Systems, 1995, ISBN 0-471-12094-4.

[Bishop 2002] Bishop, P.G., Rescaling Reliability Bounds for a New Operational Profile, Centre for Software Reliability and Adelard
City University, London, UK.

[Blackburn 1998] Blackburn, M.R., and J. Fontaine. *Specification Transformation to Support Automated Testing*, SPC-97037-MC, version 01.00.02. Herndon, Virginia: Software Productivity Consortium, 1998.

[Blackburn 2003] Blackburn, M.R., R.D. Busser, A.M. Nauman, Interface-Driven, Model-Based Test Automation, CrossTalk, May 2003. http://www.stsc.hill.af.mil/crosstalk/2003/05/Blackburn.html

[Blackburn 2004a] Blackburn, M.R. *Guidance for Achieving Mission Assurance in Software-Intensive Systems*, SPC-2004041-MC, version 1.0. Herndon, Virginia: Software Productivity Consortium, 2004.

[Blackburn 2004b] Blackburn, M.R., R.D. Busser, A.M. Nauman, T.R. Morgan, Using Models for Development and Verification of High Integrity Systems, INCOSE MARC 2004, the INCOSE Mid-Atlantic Regional Conference • November 2–4, 2004 • Arlington, Virginia. http://www.software.org/pub/externalpapers/papers/blackburn-2004-1.pdf

[Caseley 2003] Caseley, P.R., Sqn Ldr N. Tudor, Dr. C. O'Halloran, DSTL, RAF, Qunetiq. *The Case For An Evidence Based Approach To Software Certification*. Crown, 2003.

[DeMillo 1978] DeMillo, R. A., W. M. McCracken, R. J. Martin, and J. F. Passafiume. *Software Testing and Evaluation*. Benjamin/Cummings Publishing Company, Redwood City, CA 1978.

[Frankl 1998] Frankl, P. G.; Hamlet, R. G.; Littlewood, B.; Strigini, L.: Evaluating Testing Methods by Delivered Reliability, IEEE Trans. Software Eng. 24 (1998), pp. 587 -601

[Gokhale 1996] Gokhale, S., P. Marinos, and K. Trivedi. "Important Milestones in Software Reliability Modeling." *Communications in Reliability Maintainability and Serviceability*, An International Journal published by SAE International, 1996.

[Grottke 2001] Grottke, M., K. Dussa-Zieger, Systematic vs. Operational Testing: The Necessity for Different Failure Models, Proc. 5th Conference on Quality Engineering in Software Technology, Nuremberg, 2001.

[Hamlet 1977] Hamlet, R. G. "Testing Programs with the Aid of a Compiler." *IEEE Transactions on Software Engineering* (July 1977).

[Hayhurst 2001] Hayhurst, Kelly J., Dan S. Veerhusen, John J. Chilenski, and Leanna K. Rierson. *A Practical Tutorial on Modified Condition/Decision Coverage*, NASA/TM-2001-210876, 2001. http://techreports.larc.nasa.gov/ltrs/PDF/2001/tm/NASA-2001-tm210876.pdf

[Heitmeyer 1996] Heitmeyer, C., R. Jeffords, B. Labaw, Automated Consistency Checking of Requirements Specifications. ACM TOSEM, 5(3):231-261, 1996. See http://chacs.nrl.navy.mil/personnel/heitmeyer.html.

[Howden 1976] Howden, W.E. "Reliability of the Path Analysis Testing Strategy." IEEE Transactions on Software Engineering 2,9(1976): 208-215.

[Howden 1980] Howden, W.E. "Functional Program Testing." *IEEE Transactions on Software Engineering* 6,2(1980): 162-169.

[IEEE 1988] *Guide for the Use of IEEE Standard Dictionary of Measures to Produce Reliable Software*, Institute of Electrical and Electronics Engineers, IEEE 981.2-1988.

[Laprie 1984] Laprie, J.C., “Dependability Evaluation of Software Systems in Operation.” *IEEE Transactions on Software Engineering* SE-10 (1984): 701-714.

[Lyu 1995] Lyu, Michael R. *Software Fault Tolerance*. John Wiley & Sons Ltd.: New York, New York, 1995.

[McCabe 2002] McCabe, R., M. R., Blackburn, Applying the Test Automation Framework With Use Cases and the Unified Modeling Language, SPC-2002048-MC, version 1.0. Herndon, Virginia: Software Productivity Consortium, 2002.

[Musa 1993] Musa, J. D. “Operational Profiles in Software-Reliability Engineering.” *IEEE Software* 10,2 (March 1993).

[Offut 1994] Offutt, A. J., A. Lee, G. Rothermel, R. Untch, and G. Zapf. An *Experimental Determination of Sufficient Mutant Operators*, Internal Draft. Fairfax, Virginia: George Mason University, October 1994.

[Poore 1998] Poore, J. H., and Carmen J. Trammell. Software Engineering Technology, Inc., 1998.
www.stsc.hill.af.mil/crosstalk/1998/04/statistical.pdf

[Richardson 1992] Richardson, D.J., S. Leif Aha, and T.O. O'Malley. “Specification-Based Oracles for Reactive Systems.” In *Proceedings, 14th International Conference on Software Engineering*. New York, NY pages 105-118,1992.

[RTCA 1992] Radio Technical Corporation for Aeronautics Special Committee 167. *Software Considerations in Airborne Systems and Equipment Certification*, DO-178B/ED-12B. Washington, DC: Radio Technical Corporation for Aeronautics Special Committee, December 1992.

[Schneidewind 2004] Schneidewind, N.F., A Recommended Practice for Software Reliability, CrossTalk, August, 2004.
http://www.stsc.hill.af.mil/crosstalk/2004/08/0408Schneidewind.html

[Storey 1996] Storey, N., Safety-Critical Computer Systems. Harlow, England: Addison-Wesley, 1996.

[Tsai 1990] Tsai, W.T., D. Volovik, and T.F. Keefe. “Automated Test Case Generation for Programs Specified by Relational Algebra Queries.” *IEEE Transactions on Software Engineering* 16,3 (March 1990):316-324.

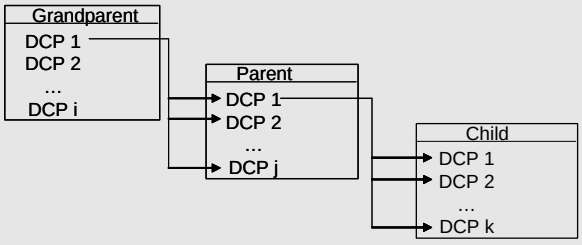
[White 1980] White, L. J., and E. I. Cohen. “A Domain Strategy for Computer Program Testing.” *IEEE Transactions on Software Engineering* SE6,3 (May 1980).

History and Technical Details of TAF

The core capabilities of this approach were developed in the late 1980s and proven through use in support of Federal Aviation Administration (FAA) certifications for flight-critical avionics systems. The approach supports requirement-based test coverage mandated by the FAA with significant life-cycle cost savings. The approach and tools have been used for modeling and testing system, software integration, software unit, and hardware/software integration functionality. It has been applied to critical applications in medical and aerospace, supporting automated test-driver generation in most languages (e.g., C, C++, Java, Ada, Perl, PL/1, Structured Query Language (SQL), as well as proprietary languages, and test environments.

TAF model translators convert various modeling notations into a T-VEC specification [Blackburn 1998]. T-VEC compiles the specification and derives the partitions for all subdomains associated with the modeled functionality. T-VEC selects test data for subdomains of an input space based on the constraints of the model that define the subdomain. A set of test vectors is generated for each logically AND'ed set of constraints, referred to as the DCPs. The underlying specification language of the TAF test generator has continually evolved to support an extensive set of logical and mathematical operators that extend the standard arithmetic operators (e.g., trigonometric, intrinsic, integrators, quantization, matrix operations). The modeling language uses functions or other forms of model references to support model composition that is required to scale to large and complex applications. Test generation support is provided for models that are composed hierarchically or sequentially.

T-VEC supports model checking. Model checking is a "lightweight" form of a formal method that checks the truth or falsity of modeled specifications for each DCP. A simple example is a logical contradiction, where (x>0) & (x<0). TAF performs model checking on simple models as well as hierarchically composed models and generates test vectors as a by-product. Test selection for higher-level subsystems typically depends on constraints or functions of lower-level subsystems. References from higher-level to lower-level subsystems must be supported by at least one DCP in a lower-level subsystem. If there is no DCP thread from a higher-level subsystem to a lower-level subsystem, this proves that there is no input space associated with the model (i.e., the input space for the DCP is null). When generating test vectors, the inputs are selected from the inputs, but if the input space is null, no tests can be selected; this is an invalid specification within the model.



This model-checking capability also supports proof-of-safety properties. Model assertions representing safety properties can be specified. During the test generation process, if test vectors are generated from a safety property that is associated with a model, the test vector identifies a DCP thread through the model, where the safety property is violated. Other checks, such as mathematical errors or potential errors (e.g., division by a domain that spans zero) are flagged as being a potential divide-by-zero hazard, or range overflow/underflow (i.e., variables which at some point in the mode have values outside the specified bounds of the type of that variable).

About the Systems and Software Consortium, Inc.

The Systems and Software Consortium, Inc. (SSCI) enables its members to solve complex challenges on large, software-intensive, network-centric systems. SSCI provides leadership and business-advancing intelligence on standards and trends, to help our members enhance their business performance.

SSCI is a world-class leader in the application of tools and processes to support software and systems engineering and design management, integration, and mission assurance.

About the Particular Program

Members with general questions or comments on any of the topics in this paper or related topics, or members interested in applying TAF with SSCI assistance, should contact the author or their member account director (see http://www.systemsandsoftware.org/pub/keycontacts.asp).

For more on TAF, see SSCI's TAF website at http://www.systemsandsoftware.org/pub/taf/testing.html

SSCI is interested in your comments and suggestions.
Please send your thoughts and insights to
blackburn@systemsandsoftware.org.