

# Interaction Testing in Model-Based Development: Effect on Model-Coverage

Renée C. Bryce  
Computer Science  
Univ. of Nevada - Las Vegas  
Las Vegas, NV 89154-4019  
reneebruce@cs.unlv.edu

Ajitha Rajan  
Comp. Sci. and Eng.  
University of Minnesota  
Minneapolis, MN 55455  
arajan@cs.umn.edu

Mats P.E. Heimdahl  
Comp. Sci. and Eng.  
University of Minnesota  
Minneapolis, MN 55455  
heimdahl@cs.umn.edu

## Abstract

*Model-based software development is gaining interest in domains such as avionics, space, and automotives. The model serves as the central artifact for the development efforts (such as, code generation), therefore, it is crucial that the model be extensively validated. Automatic generation of interaction test suites is a candidate for partial automation of this model validation task. Interaction testing is a combinatorial approach that systematically tests all t-way combinations of inputs for a system. In this paper, we report how well interaction test suites (2-way through 5-way interaction test suites) structurally cover a model of the mode-logic of a flight guidance system. We conducted experiments to (1) compare the coverage achieved with interaction test suites to that of randomly generated tests and (2) determine if interaction test suites improve the coverage of black-box test suites derived from system requirements. The experiments show that the interaction test suites provide little benefit over the randomly generated tests and do not improve coverage of the requirements-based tests. These findings raise questions on the application of interaction testing in this domain.*

## 1. Introduction

In model-based development, the development effort is centered around a model of the proposed software system. The model also serves as the fundamental artifact for validation and verification purposes, where it can be subjected to various types of analysis, for example, model checking and theorem proving. Given the central role of the model, it is crucial to adequately validate it to ensure that it meets the users' needs; a natural choice for the majority of these validation efforts is blackbox testing.

Blackbox tests derived from requirements exercise the behavior of software (models or implementations), without bias to its internal structure. Partial automation of black-

Globally Async. Switch : {On, Off}  
Transfer Switch: {On,Off}  
Overspeed: {True, False}  
Nav. Switch: {On, Off}  
Autopilot Engage Switch: {On, Off}

**Table 1. Example input parameters**

Globally Async. Switch	Transfer Switch	Overspeed	Nav. Switch	Autopilot Engage Switch
On	On	True	Off	Off
Off	On	False	On	On
On	Off	True	On	On
Off	Off	False	Off	Off
Off	Off	True	Off	On
On	On	False	On	Off

**Table 2. Example pair-wise interaction test suite**

box testing is highly desirable to realize the full potential of model-based development. One easily mechanized approach to blackbox testing is to systematically test combinations of input parameters [9]; *software interaction testing* is one such strategy. Software interaction testing is a combinatorial approach that systematically tests all t-way combinations of inputs for a system [6]. For instance, consider a subset of the inputs to a Flight Guidance System (FGS) shown in Table 1. There are five input parameters (variables) that have two possible settings each. To exhaustively test all possible combinations of these input parameters requires  $2^5 = 32$  tests, whereas, an interaction test suite that covers all pairwise (2-way) combinations of input parameters only requires 6 tests, as shown in Table 2. The difference in the number of test suites is more dramatic for the complete FGS, which includes 40 parameters that have two

possible settings each. Testing all combinations of input parameters for the FGS requires  $2^{40} = 1,099,511,627,776$  tests, whereas pairwise interaction testing can be done in 13 tests, as seen in the forthcoming experiments.

Interaction testing is appealing since generation of test suites can be automated (see [6] for survey of algorithms that generate interaction test suites). There are also indications that interaction testing can be an effective test selection technique. For instance, a study of a web browser shows that 70% of reported problems were identifiable with 2-way interaction testing, approximately 90% with 3-way interaction testing, and 95% with 4-way interaction testing [12]. Another example shows that 97% of defects in 109 medical devices were revealed due to pairwise testing of parameter settings [13]. Although these results are impressive, these studies investigated existing problem reports and hypothesize that these problems would have been detected had interaction testing been used during development; these results were not achieved in an actual testing project. Other studies that apply interaction testing directly to implementations show that it may provide high code coverage [8, 4]; and it may be used to localize [7] and to characterize faults [21] in a small number of tests.

To evaluate the applicability of interaction testing in the model-based domain we are conducting a series of experiments. In this paper, we set out to investigate how well 2-way through 5-way interaction test suites cover the structure of a model for a Flight Guidance System (FGS) developed at Rockwell Collins<sup>1</sup>. (In the avionics domain, structural coverage is important because of regulatory concerns.) We report on two experiments: (1) a comparison of the structural coverage achieved with interaction tests with that of randomly generated tests and (2) an attempt to determine if adding interaction test suites improves the coverage of black-box test suites derived from the system requirements.

In the next section we introduce our experiment. We discuss the results and point to future research directions in Section 3. Section 4 covers the internal and external threats to the validity of our study; we provide conclusions in Section 5.

## 2. Experiment

In this initial evaluation of the applicability of interaction testing in the model-based domain, we formulate two hypotheses for our study.

**Hypothesis 1:** Interaction test suites achieve higher structural coverage of a model than randomly generated test suites of the same size.

<sup>1</sup>We thank Dr. Michael Whalen, Dr. Steve Miller, and Dr. Alan Tribble of Rockwell Collins Inc. for letting us use the models they have developed during our collaboration.

**Hypothesis 2:** Augmenting a requirements-based black-box test suite with interaction test suites will increase model-coverage.

The model under test (described in more detail in Section 2.1) is expressed in a state-based language with characteristics similar to Statecharts [10] and Stateflow [15]. To measure model-coverage we select a collection of model coverage measures that can be intuitively defined over such models; we measure state coverage, decision coverage, transition coverage, and MC/DC (discussed further in Section 2.2.4).

### 2.1. The Case Study: a Flight Guidance System

To provide a realistic evaluation of interaction testing in the model-based domain we use an example from commercial avionics—a close to production model of a Flight Guidance System (FGS). The Flight Guidance System used here is a component of an overall Flight Control System in a commercial aircraft. It compares the measured state of an aircraft (position, speed, and altitude) to the desired state and generates pitch and roll-guidance commands to minimize the difference between the measured and desired state. The FGS consists of the mode logic that determines which lateral and vertical modes of operation are active and armed at any given time, and the flight control laws that accept information about the aircraft's current and desired state to compute the pitch and roll guidance commands. In this paper we use the mode logic of the FGS [16].

The FGS was built using RSML<sup>-e</sup> [18, 20] notation — RSML<sup>-e</sup> is based on the Statecharts [10] like language Requirements State Machine Language (RSML) [14]. RSML<sup>-e</sup> is a fully formal and synchronous data-flow language without any internal broadcast events (the absence of events is indicated by the <sup>-e</sup>).

An RSML<sup>-e</sup> specification consists of a collection of input variables, state variables, input/output interfaces, functions, macros, and constants. *Input variables* are used to record the values observed in the environment. *State variables* are organized in a hierarchical fashion and are used to model various states of the control model. The state variables change state when the guard condition on a state-transition is satisfied.

There are 40 input variables (also referred to as input parameters) associated with the FGS. Each variable only takes on two possible values (Boolean or two enumerations). As an alternative to the  $2^{40} = 1,099,511,627,776$  combinations needed for exhaustive testing of all possible inputs, we apply interaction testing to systematically tests all  $t$ -way interactions in the system with test suites of feasible size.

## 2.2. Experimental Setup

We conduct the experiment through the steps outlined below, with each step elaborated in the following sections.

1. We generate three sets of test suites for comparison: (1) a set of interaction test suites ranging from 2-way through 5-way coverage of the inputs of the FGS, (2) a collection of random test suites, and (3) two sets of requirements-based tests derived from the English language requirements of the FGS.
2. We append the interaction test suites to the requirements-based tests to yield a collection of augmented requirements-based test suites.
3. We run the test suites on the RSML<sup>-e</sup> FGS model and measure the coverage achieved.

In the remainder of this paper we provide a detailed description of activities involved in the experiment and discuss our findings.

### 2.2.1 Interaction test suite generation

The interaction test suite generation algorithm constructs test suites one-test-at-a-time until all  $t$ -tuples are covered [3]. For each test, the algorithm assigns values to the input parameters one-at-a-time in random order. We refer to input parameters that have been assigned values as *fixed*, those not yet assigned settings are called *free*. To select a setting for a free parameter, the algorithm selects the setting that covers the largest number of previously uncovered  $t$ -way combinations in relation to the fixed parameters. In the case where two or more assignments cover the same number of  $t$ -way combinations we randomly select one of the assignments. Once a test is complete, 100 iterations of hill climbing permute settings for parameters in a test since this sometimes increases the number of  $t$ -tuples in incremental tests; and consequently reduces the overall size of our test suites here. (This implementation of hill climbing is a variation of heuristic search applied in [5], however we apply search to incremental tests rather than to an entire test suite.) The algorithm generates and appends new tests to the test suite until all  $t$ -tuples are covered. The interaction test suites generated provide 2-way, 3-way, 4-way, and 5-way coverage of the inputs of the FGS in 13, 33, 83, and 214 tests respectively.

### 2.2.2 Random test suite generation

For each interaction test suite, we generate a random test suite of the same size. We do this by randomly assigning settings to each of the 40 FGS input variables, with the only restriction that duplicate tests are not permitted in the resultant test suite.

“If the onside Flight Director cues are off, the onside Flight Director cues shall be displayed when the Auto-Pilot is engaged.”

(a)

$$G((\neg \text{Onside\_FD\_On} \wedge \neg \text{Is\_AP\_Engaged}) \rightarrow X(\text{Is\_AP\_Engaged} \rightarrow \text{Onside\_FD\_On}))$$

(b)

**Table 3. (a) Sample high-level requirement on the FGS (b) LTL property for the requirement**

### 2.2.3 Requirements-based test suite generation

In addition to the formal RSML<sup>-e</sup> model of the FGS, our case example also consists of 293 informal requirements describing the intended behavior of the FGS. In a previous investigation we formalize these requirements as Linear Temporal Logic (LTL) properties [17] and devised a technique to automatically generate black-box tests that provide structural coverage of these LTL properties [19]. These formalized requirements serve as the basis for the generation of two sets of blackbox tests; one set providing *Requirements Antecedent Coverage* and a second set providing *Unique First Cause Coverage* (both discussed in detail below).

**Requirements Antecedent Coverage:** At a minimum, a blackbox test suite should contain one test per requirement to illustrate one way in which the requirement can be met. As an example, consider the FGS requirement in Table 3. (In the formalization of the requirement, G means that the requirement applies universally and the X means “in the next state”.)

A test derived from the informal requirements might look like the scenario in Table 4. Alternatively, we could simply leave the auto-pilot turned off and it does not matter what happens to the flight director. Technically, this test also demonstrates one way in which the requirement is met, but the test is not particularly illuminating. When automating the generation of tests from LTL properties we do not want to generate tests that illustrate the satisfaction of requirements in trivial ways.

1. Turn the Onside FD off
2. Disengage the AP
3. Engage the AP
4. Verify that the Onside FD comes on

**Table 4. Manually developed requirements-based test scenario**

Therefore, we define a requirements coverage metric—

*requirements antecedent coverage*—that exercises the antecedent in such requirements. A test providing requirements antecedent coverage over the requirement in Table 3 ensures that the antecedent becomes true at least once (the flight director must be turned off and the auto-pilot disengaged). We use this requirements antecedent coverage to derive one test per requirement yielding a test suite of 293 tests.

**Unique First Cause Coverage:** Most of the requirements in the FGS are of the form  $G(A \rightarrow B)$  (as the example in Table 3). Nevertheless, the requirements are not always that simple; the conditions making up  $A$  and  $B$  in the requirement can be quite complex. Rather than simply requiring that  $A$  is true in the test, it can be desirable to derive tests that demonstrate the effect of *each* atomic condition making up the complex conditions in the requirement; requirements antecedent coverage would not be able to do this. Therefore, we define a coverage metric called *Unique First Cause (UFC)* coverage over requirements [19]. A test suite is said to satisfy UFC coverage over a set of LTL formulas if executing the tests in the test suite guarantees that:

- every basic condition in a formula has taken on all possible outcomes at least once
- each basic condition has been shown to independently affect the formula's outcome.

Given this stronger notion of requirements coverage, one can generate a second blackbox test suite from the 293 requirements and generate a test suite with UFC coverage in 713 tests.

**A note on the blackbox test suites:** In our work on requirements-based testing we found that the test suites providing requirements antecedent coverage and UFC coverage of the FGS requirements provide reasonable, but not high, coverage over the RSML<sup>-e</sup> FGS model. The reason being that the set of requirements for the FGS is incomplete; a substantial number of the requirements were defined using macros (abbreviations) that abstracted away numerous complex conditions. Additional requirements defining these macros were not captured when the requirements were formalized as LTL properties. *Therefore, the requirements-based test suites used in this study are good, but they are not quite up to the standards one would expect from a thorough blackbox testing effort.* The implication of this observation will be discussed in Section 3.

#### 2.2.4 Model Coverage Measurement

The execution environment we use (called NIMBUS) includes a tool to measure different kinds of coverage

achieved over RSML<sup>-e</sup> models [11]. The tool measures the state, transition, decision, and MC/DC coverage achieved from executing test suites. The different measures of coverage are informally defined as follows:

**State Coverage:** requires that the test suite has tests that force each state variable defined in the model to take on all possible values at least once.

**Transition Coverage:** (analogous to branch coverage in code) requires that the test suite has tests that exercise every transition in the model at least once.

**Decision Coverage:** each decision occurring in the model evaluates to true at some point in some test and evaluates to false at some point in some other test. A decision in this context is defined as any complex boolean expression. (An example of a *decision* is  $A \text{ or } B$ , where  $A$  and  $B$  are both *conditions*).

**Modified Condition and Decision Coverage (MC/DC):** requires us to show that (1) every condition within the decision has taken on all possible outcomes at least once, and (2) every condition is shown to independently affect the outcome of the decision.

### 3. Experimental Results and Discussion

**Hypothesis 1:** Interaction test suites achieve higher structural coverage of a model than randomly generated test suites of the same size.

The results of running our four interaction test suites (one each that covers interactions of size  $t=\{2,3,4,5\}$ ) and the randomly generated tests of the same size can be seen in Table 5. The results show that the interaction tests perform only slightly better than the random tests for all four types of coverage. The difference is most striking when fewer tests are executed, such as with  $t=2$  test suites. Thus, Hypothesis 1 is weakly supported. This is not surprising since random tests are likely to cover many  $t$ -tuples. Table 6 shows that in our experiment, each of the random test suites cover at least 77% of the  $t$ -tuples in each case. This is similar to results in [1] where a random test suite covers 88% of pairs that a pairwise interaction test suite of the same size does.

In the second experiment, the requirements-based tests are augmented with  $t=\{2, 3, 4, 5\}$  interaction tests to determine if the coverage of the blackbox suites is improved.

**Hypothesis 2:** Augmenting a requirements-based blackbox test suite with interaction test suites will increase model-coverage.

Table 7 shows that the interaction test suites do not increase coverage of the model in any significant way for either blackbox test suite. For the antecedent coverage test

	Random	2-way	Random	3-way	Random	4-way	Random	5-way
Number of tests	13	13	33	33	83	83	214	214
State cov.	77.4%	84.8%	78.7%	87.2%	84.2%	90.9%	90.2%	90.9%
Transition cov.	57.6%	64.0%	58.7%	65.7%	64.0%	68.0%	67.4%	68.0%
Decision cov.	71.4%	74.0%	72.6%	75.3%	74.8%	77.6%	78.4%	79.1%
MCDC cov.	7.4%	10.3%	8.1%	11.0%	10.0%	12.3%	11.9%	12.3%

**Table 5. Model coverage achieved on the FGS with randomly generated tests and interaction tests.**

	No. of rows	% of $t$ -tuples covered
$t=2$	13	77.2
$t=3$	33	77.4
$t=4$	83	82.6
$t=5$	214	87.8

**Table 6. Percentage of  $t$ -tuples covered in the random test suites.**

	Ant. No. of tests	Ant. % $t$ -tuples covered	UFC No. of tests	UFC % $t$ -tuples covered
$t=2$	291	99.9	713	99.8
$t=3$	291	91.3	713	95.7
$t=4$	291	63.1	713	80.2
$t=5$	291	42.0	713	60.7

**Table 8. Percentage of  $t$ -tuples covered in the requirements-based test suites.**

suite, the interaction tests might find an occasional test that increases the coverage a fraction of a percent. The interaction tests do not increase coverage when used to augment the Unique First Clause (UFC) test suites. Therefore, our second hypothesis is not supported; interaction tests do not improve the coverage of existing blackbox test suites. This result is somewhat disappointing since we know that the requirements-based blackbox test suites are deficient (as discussed in Section 2) and there is scope for coverage improvement.

We believe the reason for the poor coverage achieved by the interaction test suites can be attributed to the nature of the FGS system. Much of the behavior of the FGS is dependent on *state* that is computed and stored over two or more execution steps. To reach many of the states in the system, a sequence of steps is required to set the state appropriately. Achieving high structural coverage over the model may therefore require reaching some of these “deeper” states. To illustrate this, consider the following decision in the FGS model, where *PREV\_STEP* refers to the value of a variable in previous step:

$$\begin{aligned}
 &(PREV\_STEP(Independent\_Mode) = Off) \wedge \\
 &\quad \neg(Pilot\_Flying = This\_Side) \wedge \\
 &\quad (Offside\_Modes = On)
 \end{aligned}$$

To exercise the above decision in the model up to MC/DC coverage, we would have to be able to set values (*Off*, *On*, and *Undefined*) for the *Independent\_Mode* input variable in the previous step. Therefore, tests that provide MC/DC coverage of the above decision need to be at least two steps long, so as to allow the value of *Independent\_Mode* in the previous step to be set. Many of the decisions in the FGS model are of the above form.

The interaction tests used here are only one step long, so it is impossible for them to achieve good coverage over the model when the coverage criteria requires tests to reach states that are “deeper” in the reachable state space. For example, no set of interaction tests can achieve MC/DC coverage of the above decision in the FGS model (since it would have to be at least two steps long). We hypothesize that missing state information in the interaction test suites is a significant factor in the poor coverage recorded for interaction testing. Given the nature of software, we believe that this problem in interaction testing (missing state information) generalizes to many other software systems. In future work, we will consider state information when generating interaction test suites, both by treating state information in the combinations that must be satisfied to achieve adequate coverage and by generating longer tests over the input space. We believe that interaction test suites generated this way will record much better model coverage, and fault finding. We plan to evaluate these hypotheses in our future work.

While the requirements-based test procedures generate sequences of steps that enable visitation of states deeper within the state space and provide better structural coverage of the model than the interaction tests, they also provide good coverage given the 2-way and 3-way interaction criteria, however they do not exercise many of the 4-way and 5-way input interactions. This is shown in Table 8. Intuitively, since the requirements-based tests do not include a large number of 4-way and 5-way interactions, it is possible that the requirements-based tests can not reveal some faults that can be uncovered by the 4-way and 5-way interaction

	Antecedent	Augmented w/ 2-way	Augmented w/ 3-way	Augmented w/ 4-way	Augmented w/ 5-way
Number of tests	293 tests	314	324	374	505
State cov.	98.8%	98.8%	98.8%	98.8%	98.8%
Transition cov.	89.5%	90.1%	89.5%	90.1%	90.1%
Decision cov.	88.6%	88.9%	88.6%	88.9%	88.9%
MCDC cov.	23.9%	24.2%	23.9%	24.2%	24.2%
	UFC	Augmented w/ 2-way	Augmented w/ 3-way	Augmented w/ 4-way	Augmented w/ 5-way
Number of tests	713 tests	726	746	796	927
State cov.	99.1%	99.1%	99.1%	99.1%	99.1%
Transition cov.	99.4%	99.4%	99.4%	99.4%	99.4%
Decision cov.	83.9%	83.9%	83.9%	83.9%	83.9%
MCDC cov.	53.5%	53.5%	53.5%	53.5%	53.5%

**Table 7. Model Coverage comparisons of requirements-based test suites augmented with 2-way through 5-way interaction tests**

test suites. Therefore, although the interaction tests did not help to improve the coverage of the model, the augmented tests may be of help when it comes to fault finding. We plan to investigate this issue further by running experiments that compare the fault finding capability of the original and augmented requirements-based test suites.

One additional consideration is that the requirements-based test suites were generated in a *constrained environment* that disallowed certain combinations of inputs (based on environmental assumptions for the model), while the interaction test suites did not have these constraints. This difference brings up concerns. First, the interaction test suites may have been constructed of different size in order to address constraints of combinations that cannot be combined together. Indeed, work in [2] shows that constraints can result in smaller or larger sized test suites. Second, in the practical application of the study here, the difference in the number of tests executed and the states visited may yield different results. Our intuition is that the model coverage from interaction testing will not significantly increase given the constraints used for requirements-based testing. Nevertheless, these questions involving constraints warrant future study.

#### 4. Threats to Validity

There are three obvious threats to the external validity that prevent us from generalizing our observations. First, and most seriously, we are using only one instance of a formal model in our experiment. Although the FGS is an ideal example, it was developed by an external industry group, it is large, it represents a real system, and is of real world

importance, it is still only one instance. The FGS model is entirely modelled using enumerated variables; it is impossible to generalize the results to systems that, for example, contain numeric variables and constraints.

Second, we use one algorithm to generate a test suite that covers all  $t$ -way interactions. There are many other algorithms with the same capability, but each of them may produce test suites that cover the  $t$ -way interactions in a different order and in a different number of tests.

Third, we use a single test generation technique to generate the requirement-based test suites. The test generation strategy plays an important role in test suite effectiveness (with respect to model coverage and fault finding). We need to evaluate the effect of other test generation strategies on test suite effectiveness.

Although there are several threats to the external validity, we believe the results raise serious questions about the benefits of interaction testing techniques in this domain. Further work needs to explore these concerns.

#### 5. Conclusions

We conducted experiments to measure the structural coverage of 2-way through 5-way interaction test suites applied to the mode-logic of a flight guidance system model. The results show that the interaction test suites provide little benefit over the randomly generated tests and do not improve coverage of blackbox tests derived from the informal requirements. This raises concerns on the application of interaction testing in the model-based domain. Nevertheless, the results for model coverage can be quite different from that obtained for fault finding effectiveness. In our future

work, we will evaluate the fault finding capability of the interaction test suites over systems in this domain.

Interaction test suites applied here do not consider state information, and thereby only test combinations of input parameters over a single step. The experiments point out that for the FGS, generating single-step interaction test cases may not be sufficient, longer (multi-step) tests are needed to achieve better coverage of the model. Given the nature of software, we believe that missing state information in the interaction test suites will cause a similar problem in many other software systems. In our future work, we plan to generate interaction test suites that include state information and conduct experiments to evaluate their effectiveness for fault finding.

## 6 Acknowledgements

We would like to thank Dr. Michael Whalen from Rockwell Collins Inc. for his helpful discussions and insightful comments.

## References

- [1] J. Bach and P. Shroeder. Pairwise testing: A best practice that isn't. In *Proceedings of 22nd Pacific Northwest Software Quality Conference*, pages 180–196, 2004.
- [2] R. C. Bryce and C. J. Colbourn. Prioritized interaction testing for pairwise coverage with seeding and avoids. *Information and Software Technology Journal (IST, Elsevier)*, to appear.
- [3] R. C. Bryce, C. J. Colbourn, and M. B. Cohen. A framework of greedy methods for constructing interaction tests. In *The 27th International Conference on Software Engineering (ICSE)*, pages 146–155, May 2005.
- [4] K. Burr and W. Young. Combinatorial test techniques: Table-based automation, test generation, and code coverage. In *Proceedings of the International Conference on Software Testing Analysis and Review*, pages 503–513, October 1998.
- [5] M. B. Cohen, C. J. Colbourn, P. B. Gibbons, and W. B. Mugridge. Constructing test suites for interaction testing. In *Proceedings of the International Conference on Software Engineering (ICSE 2003)*, pages 28–48, May 2003.
- [6] C. J. Colbourn. Combinatorial aspects of covering arrays. *Le Matematiche (Catania)*, 2005.
- [7] S. R. Dalal and C. L. Mallows. Factor-covering designs for testing software. *Technometrics*, 50(3):234–243, August 1998.
- [8] S. Dunietz, W. K. Ehrlich, B. D. Szablak, C. L. Mallows, and A. Iannino. Applying design of experiments to software testing. In *Proceedings International Conference on Software Engineering (ICSE '97)*, pages 205–215, October 1997.
- [9] M. Grindal, J. Offutt, and S. Andler. Combination testing strategies: a survey. *Software Testing, Verification, and Reliability*, 15:167–199, March 2005.
- [10] D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, June 1987.
- [11] M. P. Heimdahl and G. Devaraj. Test-suite reduction for model based tests: Effects on test quality and implications for testing. In *Proceedings of the 19th IEEE International Conference on Automated Software Engineering (ASE)*, Linz, Austria, September 2004.
- [12] D. Kuhn and M. Reilly. An investigation of the applicability of design of experiments to software testing. In *Proceedings 27th Annual NASA Goddard/IEEE Software Engineering Workshop*, pages 91–95, October 2002.
- [13] D. R. Kuhn, D. R. Wallace, and A. M. Gallo. Software fault interactions and implications for software testing. *IEEE Transactions on Software Engineering*, 30(6):418–421, October 2004.
- [14] N. Leveson, M. Heimdahl, H. Hildreth, and J. Reese. Requirements Specification for Process-Control Systems. *IEEE Transactions on Software Engineering*, 20(9):684–706, September 1994.
- [15] I. Mathworks. Mathworks product descriptions. <http://www.mathworks.com/>, 2005.
- [16] S. Miller, A. Tribble, T. Carlson, and E. J. Danielson. Flight guidance system requirements specification. Technical Report CR-2003-212426, NASA, June 2003.
- [17] S. P. Miller, M. P. Heimdahl, and A. Tribble. Proving the shalls. In *Proceedings of FM 2003: the 12th International FME Symposium*, September 2003.
- [18] J. M. Thompson, M. P. Heimdahl, and S. P. Miller. Specification based prototyping for embedded systems. Technical Report TR 99-006, University of Minnesota, Department of Computer Science, Minneapolis, MN, 1999.
- [19] M. Whalen, A. Rajan, M. Heimdahl, and S. Miller. Coverage metrics for requirements-based testing. In *Proceedings of International Symposium on Software Testing and Analysis*, July 2006.
- [20] M. W. Whalen. A formal semantics for  $RSML^{-e}$ . Master's thesis, University of Minnesota, May 2000.
- [21] C. Yilmaz, M. B. Cohen, and A. Porter. Covering arrays for efficient fault characterization in complex configuration spaces. In *International Symposium on Software Testing and Analysis*, pages 45–54, July 2004.