Reducing Verification Costs through Practical Formal Methods: A Survey

Mark R. Blackburn, Ph.D.
Stevens Institute of Technology

Sumit Ray
BAE Systems

Abstract

Verification of software can be as much as 88% of the total cost to deliver a high dependability system. Significant manual effort is often required to produce required verification evidence. Formal methods are believed to hold promise by providing a more automatic means of verification. Advances have been made in theorem provers and model checkers aimed to support the verification efforts, however there are several challenges in producing verification evidence when the requirement and design specifications use nonlinear and floating point constraints mixed with linear, logical, and bit constraints. Some tools apply strategies to produce tests to support verification, but the fault-finding effectiveness of the generated tests is questionable. Finally, practical means for composing formal specifications is important for constructing specifications that scale to large systems. This paper discusses the challenges, summarizes needed capabilities of formal method technologies, provides a survey of tools, with experiments to assess and compare capabilities, and finally discusses future needs to address some of the challenges.

Keyword: model-based testing, formal methods, theorem proving, test generation, constraint solving, model checking

1 Introduction

NASA presented industry data indicating that verification is 88% of the cost to produce DO-178B Level A software, and 75% for Level B software [1]. As shown in Figure 1, the DARPA META pre-program solicitation (META) describes how continually increasing complexity impacts the verification costs of software and delivery time [2]. META claims that the fundamental design, integration, and testing approaches have not changed since the 1960. The META program goal is to significantly reduce, by approximately a factor of five, the design, integration, manufacturing, and verification level of effort and time for cyber physical systems. The complexity has increased for integrated circuits, as it has for software-intensive systems, but the developers of integrated circuits have maintained a consistent level of effort for the design, integration and testing efforts, as reflected in Figure 1. The need is to understand key reasons why software-intensive systems production is different from integrated circuits. One fundamental difference is that software behavior requires nonlinear operations and constraints that are implemented on computing hardware where operations are performed and results stored in floating point representations. This makes the automated verification problem more challenging than for integrated circuits, where automated verification and analysis is based primarily on logic or bit-level manipulations. Chip developers used to rely on simulation, much like software development uses debugging and manual testing, but the chip verification would

cost more than 50% of the effort and defects that escape to the field could cost $500M[1]. They now rely more on formal methods and tools to support development and verification.

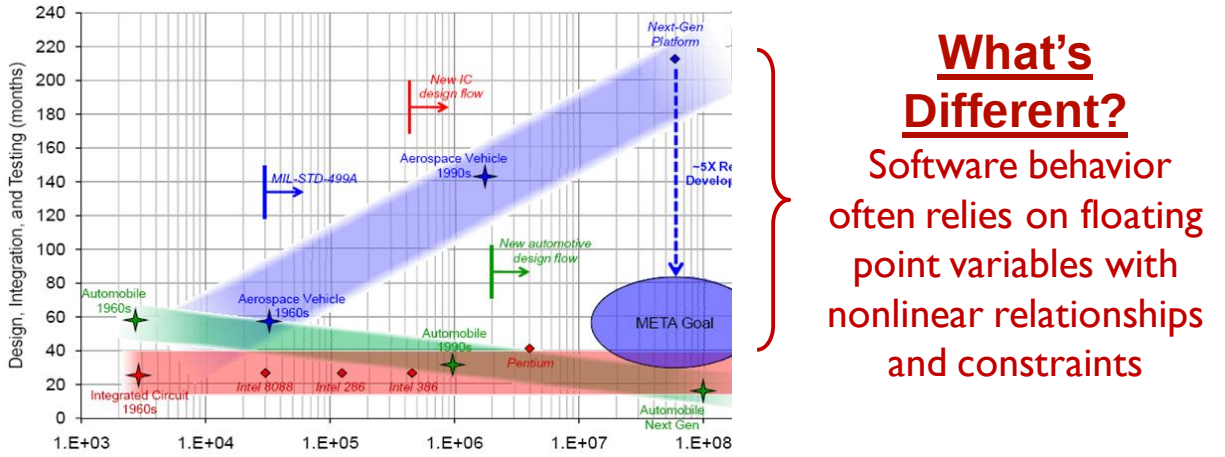


Figure 1. DARPA META Program[2]

DARPA and NASA, as reflected in Figure 2, believe that formal methods are a key technology to address the verification and validation challenges as systems become more complex. However, another issue is that our typical engineers lack advanced mathematical training and theorem proving skills needed to use formal methods to support automated analysis and test generation. Some of the specification languages are not necessarily intuitive or easy to use. The verification engines (e.g., theorem provers) do not execute like simulators or debuggers.

> Formal specifications and mathematical analysis theoretically present a way out of the dilemma posed by our inability to test even a small part of the enormous state space involved in most digital systems. They have the potential for both increasing safety and decreasing the cost of certifying flight-critical systems. The past 30 years have advanced the state of knowledge about formal methods to the point where many important problems can be solved. While formal methods are being applied to hardware in industry, the results of formal methods research for software has only rarely reached beyond the research lab and been used in industrial practice for day-to-day software development [3].

The tools have advanced, but no one tool supports all of the verification needs (e.g., logical, linear, nonlinear, and temporal) to cover the entire lifecycle. There is a need to provide practical formal methods technologies that can be applied by typical engineers of software-intensive systems. These tools must address the key gaps such as systematic analysis and verification of systems characterized by specification that include nonlinear constraint, floating point variables. Additionally, the tool-based solutions must scale to large systems.

---

[1] http://en.wikipedia.org/wiki/Pentium_FDIV_bug

[2] Modified from Paul Eremenko, META Novel Methods for Design & Verification of Complex Systems, December 22, 2009.

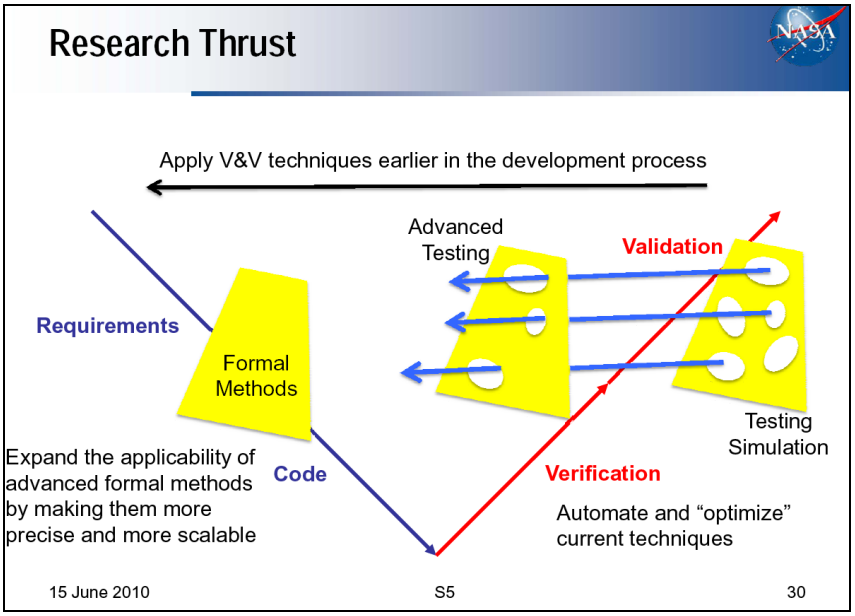


Figure 2. Early Formal Methods

1.1 Background

The DARPA Disruptive Manufacturing Technologies (DMT) initiative funded the program: Producible, Adaptive Model-based Software (PAMS) [4]. The domain-specific modeling tool chain shown in Figure 3, developed by BAE Systems and Vanderbilt University as part of PAMS illustrates a likely trend that may make formal methods more practical to apply [5]. The PAMS project was successful in developing a methodology and a suite of tools that enables rapid, model-supported adaptation. It addresses several of the previously discussed issues:

- The domain-specific modeling languages (DSML) are more intuitive to use by engineers graduating with computer and engineering degrees than some of the specification languages used by theorem provers or model checkers
- The DSML are more constrained and relate to the specific application domain, which makes them easier to learn
- Model translations were created for the Flight-Control domain-Specific Language (FCSL) that leverages tools like the T-VEC Vector Generation System (VGS) for model analysis, through theorem proving, test vector generation and test driver generation to automate the production of verification evidence
- T-VEC's VGS supports analysis (theorem proving) and test vector generation for models specified using Boolean (logical), linear, and nonlinear constraints and functions
- PAMS provides a framework for model language evolution that adapt to changing requirements throughout the software lifecycle:
    - At design time to evolve requirements and domain concepts
    - At load time to adapt components based on mission context and resource availability
    - At runtime to respond to unforeseen conditions and dynamic mission changes

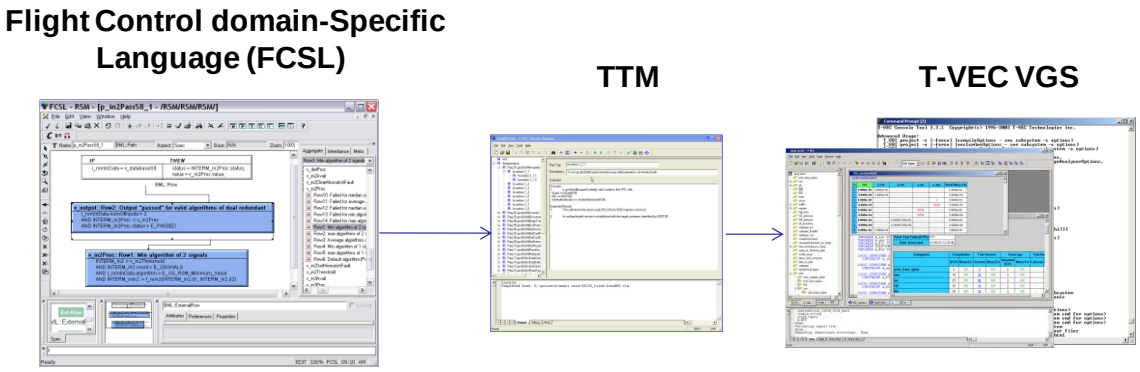


Figure 3. BAE FCSL Domain Specific Modeling Tool Chain

The FCSL tool chain demonstrated several unique capabilities, but there are some challenges that remain. Examples include the speed of the test vector generation, the fault-finding effectiveness of the selected test vectors and the effectiveness of the tool chain when applied to large scale systems. As shown in Figure 3, there are two levels of model translations, because it was easier to translate FCSL into the T-VEC Tabular Modeling (TTM) language, an XML representation of a declarative language based on the Software Cost Reduction (SCR) method and tool developed by the Naval Research Laboratory [6]. TTM has a translator to VGS. However, the two levels of model transformation introduced some issues that impacted test generation performance. While these issues were resolved, they were the catalyst for this survey. The remainder of this paper provides a survey with some comparative experiments to assess tools and emerging technologies with the goal to reduce verification costs leveraging PAMS-like technologies to provide a practical and evolvable verification environment that integrates the most promising techniques in formal analysis with automated test generation technologies.

1.2 Objectives

The paper objectives are to present a survey of technologies that provide capabilities or strategies to:

- Support analysis (e.g., theorem proving, modeling checking) to prove properties about the input specification, model or program
- Support automated test generation and associated execution that demonstrates both high fault-finding effectiveness while providing high degrees of software test code coverage
- Support verification when specifications require nonlinear, floating point constraints, in addition to linear and logical constraints across data types; this is a key difference between software and integrated circuit specifications
- Scale to large systems, where requirement and design specifications are the basis for producing the evidence used to verify the implementation in a target system software
- Be “automatic” so that it can be applied by typical engineers

1.3 Organization of Paper

Section 1 discusses issues and challenges presented by NASA and DARPA at the Safe & Secure Systems & Software Symposium, June 2010. Section 2 provides an overview of the technology survey candidates and classes. Section 3 provides a summary of surveyed technologies that are

being developed and evolved to address some of the objectives listed in Section 1.2. Section 4 discusses experiments performed with some of the surveyed technologies to assess the effectiveness of generated test data at finding faults in some experimental subjects, and assessing technologies that support analysis of specifications that are defined using nonlinear and floating point constraints. Section 5 provides ideas for future work to address gaps identified by the survey. The appendices provide additional backup material related to the experiments.

2 Comparison Overview

This section provides an overview of the survey candidates with a discussion of some of the objective and subjective measures used in the analysis. Some information comes from working directly with tools. Other information was obtained from research papers if the tool was not available for direct evaluation. The search included Satisfiability modulo theories (SMT) solvers, model checking tools, constraint solvers, and test generator tools with a particular emphasis on tool support for:

- Specification- or model-based test generation
- Model (or specification) verification, proof of properties, and the ability to find defects in models; defects are usually issues with either the requirement or design
- Test generation to support verification of an implementation on simulation, target or host
- Nonlinear, floating point constraint solving, mixed with linear and logical constraint solving across data types

Some key attributes most important to verification include:

- Ability to solve nonlinear, in addition to linear and logical, constraints
- Ability to handle nonlinear floating point constraints
- Generation of test values that are effective at finding faults
- Composition to support specification development and scalability to large systems

2.1 Candidates and Classes

The following provides a brief introduction to some of the classes of tools considered in this survey. By definition, SAT solvers deal with the satisfiability of Boolean formulae, but not linear and nonlinear constraints and functions replete in control software. Extensions, provided by SMT solvers, address formulas with linear constraints, arrays, and functions, but often have limited support for nonlinear formulas. Model-based test generation, some of which provide theorem proving capabilities that can identify anomalies within the model, produce test inputs and sometimes expected outputs. Constraint solvers were also considered as they provide capabilities to deal with nonlinear and floating point constraint solving. The candidates analyzed are shown in Figure 4. Some were not fully analyzed due both to time constraints and availability, but remain on the list as potential candidates for future consideration. In addition, some challenge problems, discussed in 4.2, helped to eliminate some candidates from detailed analysis. Finally, model checking, a type of formal methods, has provided significant capabilities for integrated circuits verification; information from a model checking survey is cited in Section 2.2.2.

Tools			Categorization/Capabilities												Constraint Solving							Data Types												Language							
Function or Suite Subset	Evaluation license	Paper Analysis Only	Interactive	Automatic	Addin Solvers	SAT	SMT	Model Checking	Convergence	Simulation	Test Generation	Test Vector Generation	Test Driver Generation	Logical	Linear (arithmetic)	Nonlinear	Trigonometric	Other (ln, log)	quatifiers	Builtins (max, floor)	Boolean	Integer	Real	Floats	Arrays	Structures	Strings	Enumerations	Tuple	User Defined	Bit vectors	SMT-Lib	Modeling	Domain-specific	Custom	Simulink	C, C# API, Java	F#
T-VEC VGS	x		x	x					x	x	x	x	x	x	x	x	x	x	x	x	x	x		x	x	x			x	x	i	x	i	x	x	p	
PVS		x	x		x	i	i	i						x	x				x		x	x	x		x									x			
Yices	x		x	x		x	x							x	x				x												x			x			
Z3	x		x	x		x	x												x		x	x	x	?	x	x		x		x	x	x			x	x	x
CVC3	x		x	x		x	x							x	x	s			x		x	x	x		x	x			x	x	x	x			x		
CalCS		x														?																						
MathSAT		x				x	x																								x						
OpenSMT		x																																			
iSAT	x		x	x		x	x	x						x	x	p					x	x		x													
HySAT (related to iSAT)		x				x	x																														
Mathworks Design Verifier	x			x	x	x	x							x	x						x	x													x		
Blast		x						x																												x	
SAL-AGT								x																														
FloPSy		x			x									x	x	p																						
RealPaver		x			x																																
CPLEX		x																																			
SpecExplorer 2010	x																																			x	x
Alloy		x		x	x														x						x							x		x			
Beaver ~related to CalCS																																							
Nmodel		x																																				
PexSolver		x																																				
Euclide		x																																				
Simplify		x																																				
DPT		x																																				

Key	
x	Support
?	Possible
i	Through integration
	None or unknown

Figure 4. Tool Candidate and Categorizations

2.2 Objective and Subjective Measures

2.2.1 SMT and Constraint Solver Measures

The SMT Competitions have been valuable for comparing SMT solvers [7]. The competition focuses on speed at which a tool can identify whether a set of constraints is satisfiable (SAT) or unsatisfiable (UNSAT) across a broad set of benchmarks. Applying this concept to models of requirements or design, or proof of properties would mean:

- If the set of constraints for a requirement model resulted in UNSAT, then there is some type of anomaly (contradiction/defect) in the requirements
- If the set of constraints applied to a safety property (e.g., weight on wheels and radar enabled) resulted in SAT, then the design model violates the safety property

The SMT competition does not address nonlinear constraint problems, because the problem is in general undecideable. However, some challenge problems have been created for nonlinear real arithmetic, and floating-point nonlinear constraint solving. Details are discussed in Section 3.

2.2.2 Model Checking Measures

A model checker uses some representation of the system model (e.g., finite state machine) as input and a temporal logic property, and then explores the state space of the system to determine if the model violates the property. If the property is violated then a counterexample is generated to illustrate the problem. The counterexample can be useful in testing. Model checking was first applied and continues to be used mostly on hardware designs.

This paper focuses on testing with model checkers in addition to proof of properties. The paper "*Testing with model checkers: A survey*" provides summary of model checking capabilities, history, and strategies for applying model checking to testing [8], [9]. One cited point to note states:

> A pilot study was conducted to investigate the suitability of condition based coverage criteria. In this experiment, test suites were generated using different condition based coverage criteria for a close to production model of a flight guidance system from Rockwell Collins, Inc. The fault detection ability of the different test suites was measured on mutant versions of the model. The experiment showed that a set of randomly generated test cases generated using the same effort were superior to all coverage based test suites [10].

2.2.3 Fault Finding Effectiveness Measures and Strategies

The final desired measure is on selecting test values that are effective at finding faults.

> There is an extensive literature on methods for generating tests that are likely, or in some cases guaranteed, to detect various kinds of hypothesized faults, but rather few of these methods have been automated. The boundary coverage method collects the various constraints implied by postconditions, preconditions, and guards, then calculates their "boundary" and selects some test points from (just) inside the boundary, and some from (just) outside [11].

It is well known that tests along domain (or subdomain) boundaries are effective in testing an entire domain [12]. The constraints of a requirement identify domain and subdomain boundaries. Figure 5 provides a simplified perspective of test input values selected at the subdomains boundaries for the preconditions (constraints) that might be associated with a requirement or design specification. The test points at the boundaries are effective at both testing for modified decision/condition coverage (MC/DC) that are associated with the paths through the code, but are effective also at finding faults. Figure 5 shows a two-dimensional space, but in general the test subspace is an n-dimensional space (polyhedron) where n is the number of input variables if the constraints are satisfiable. Ideally, one would like to select values at each of the points around the polyhedron space, but as the number of inputs grows it would take too much time to practically produce and execute the tests. Therefore MC/DC is usually the minimal test coverage criteria for high dependability software systems (e.g., aircraft, medical devices).

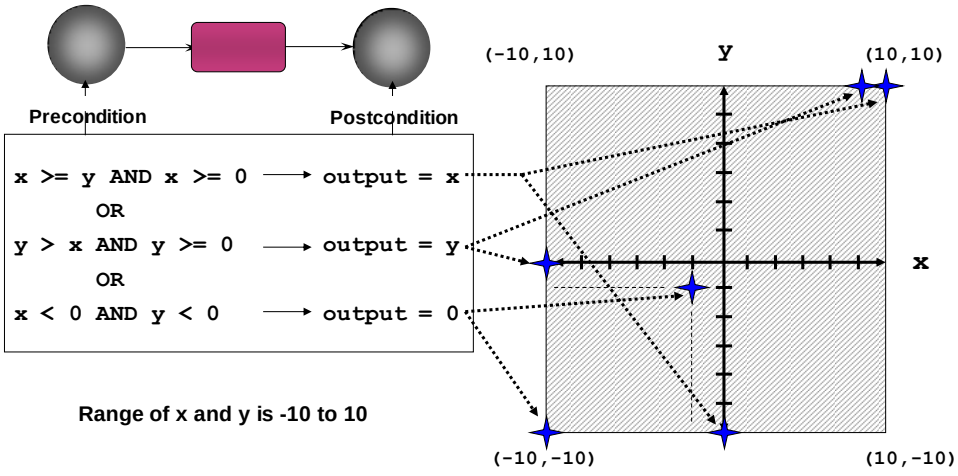
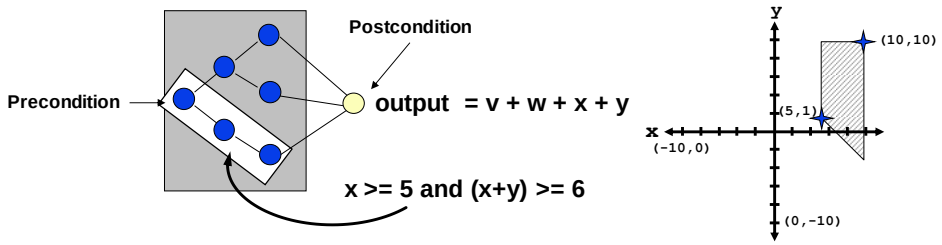


Figure 5. Test Section at the Subdomain and Domain Boundaries

Figure 6 reflects two other points about test selection strategies:

- The boundaries are not necessarily related to ground terms (variables related to constants), but may involve constrain expressions
- Selecting one boundary may not be effective at identifying faults – this is particularly relevant to testing computations, as using combinations of high-bound and low-bound values may better expose an issue with the computation or expose computational issues such as overflows or underflows



	x	y
Low-Bound Convergence		
1) Initial domain of inputs	**-10..10**	**-10..10**
2) Select minimum value for x	**5**	**-10..10**
3) Calculate minimum value of y	**5**	**1**
High-Bound Convergence		
1) Initial domain of inputs	**-10..10**	**-10..10**
2) Select maximum value for x	**10**	**-10..10**
3) Calculate maximum value of y	**10**	**10**

Figure 6. Subdomain and Test Point Selection Strategies

Another related issue, reflected in Figure 7, is addressing constraint solving and test selection, when the specification are modularized and composed into hierarchical relationships as is reflected by a simple Simulink model shown in Figure 8. Conceptually any node of a precondition might reference some subsystem, where at least one precondition/postcondition pair must satisfy the precondition of a higher-level (i.e., Parent) subsystem. Any node can be a simple relation, but can be a mathematical construct with a relation that might be nonlinear as reflected by the Simulink example in Figure 8.

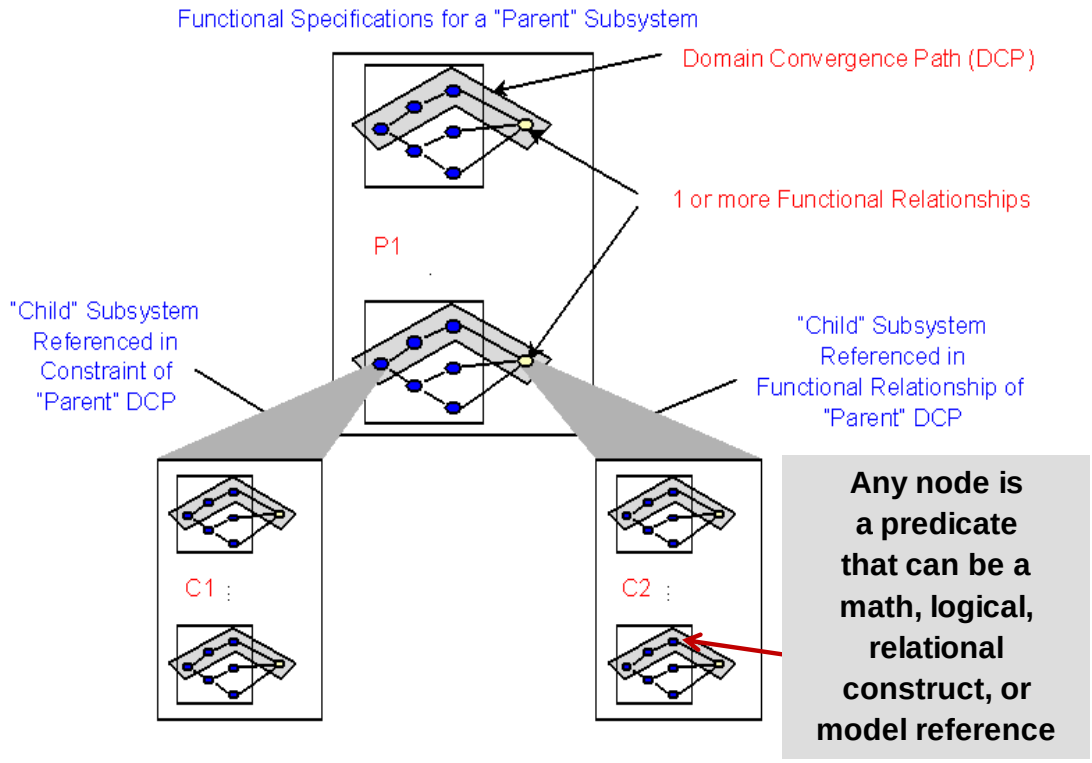


Figure 7. Constraint Solving, Test Selection and Composition

The constraints in the Simulink model shown in Figure 8 are used as one of the challenge problems for some of the tools in the survey discussed in Section 4.2. The Simulink model has two seeded defects:

- One with a linear constraint that is not satisfiable
- One with a nonlinear constraint that is not satisfiable

The nonlinear constraint shown in the red box is related to the gain operator (i.e. multiplication) in the lower-level subsystem. This creates a trivial nonlinear relationship that is represented graphically in more detail in Figure 9. The three subsystems represented by these images show the constrained space

- Subsystem relational_constraint shows the rectangular space
- Subsystem hierarchical_model_non_linear shows the nonlinear space associated with the gain operator
- Subsystem non_linear_relation at the bottom shows the overlapping domains

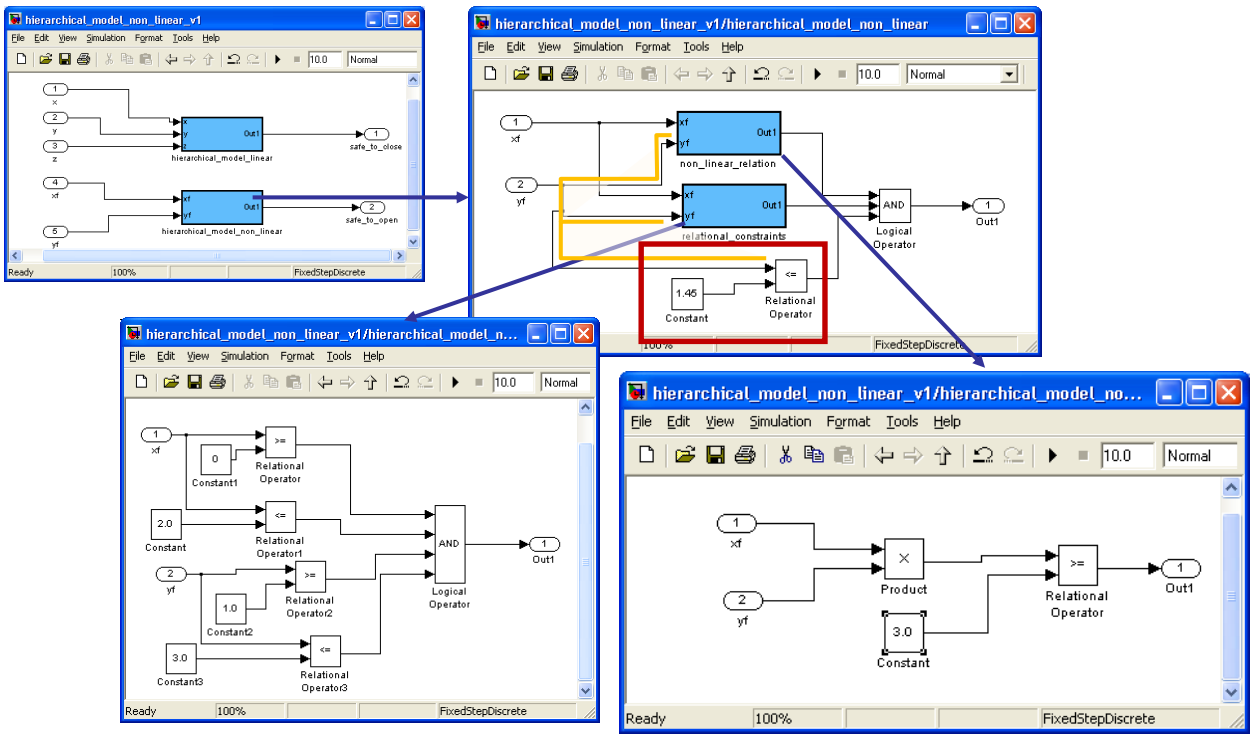


Figure 8. Simulink Example of Seeded Nonlinear Unsatisfiable Constraint

The small red line in Figure 9 shows that the constraint for the relational operator is just outside the overlapping space. This means that there is no input space that satisfies the constraints. In SMT terminology this would be UNSAT. This example illustrates a common situation where constraint issues in requirement or design models can be difficult to identify if the models are hierarchical models. Model composition is one way to address scaling to larger real-world problems, but the model analysis mechanisms must be able to identify these types of issues when models are composed hierarchically. Issues with hierarchical relationships were exposed in the SMT candidates using the experiments discussed in Section 4.2.

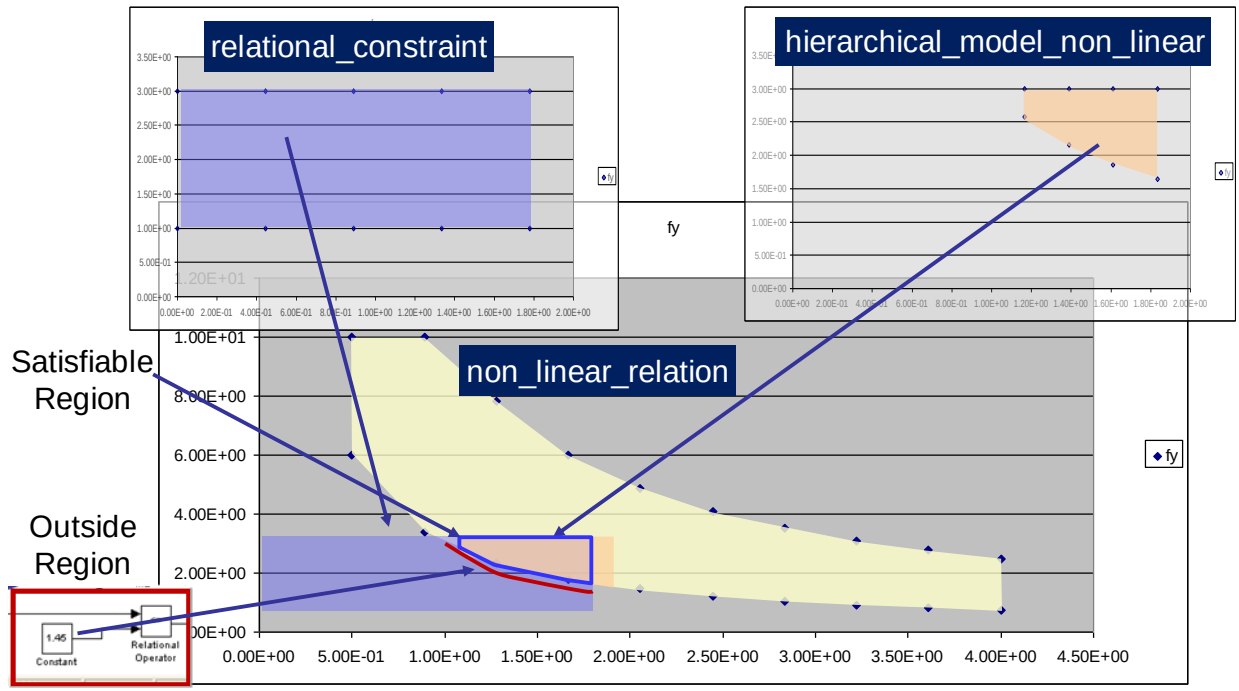


Figure 9. Representation of the Constrained Subdomains for Simulink Model

Figure 10 is a project status page that relates to a high-level verification flow chart. The model analysis data is from the model shown in Figure 9, and shows different measures related to model defects, model coverage, test result measures and test coverage measures. The typical high assurance verification process, such as DO-178B [13], would include the following checks:

1. Models should be free of defects (e.g., unsatisfiable constraints)
2. Generated tests should be executed against code that is instrumented to ensure that all paths through the code have been tested; tools such as VectorCAST and LDRA are commercial tools to measure test coverage against test criteria such as MC/DC coverage.
3. All test cases should pass (i.e., actual outputs should match expected outputs within numerical tolerances)
4. All test cases should be executed against un-instrumented code to ensure that instrumentation had no impact on the test execution results; this is the code that will be deployed in the target system.

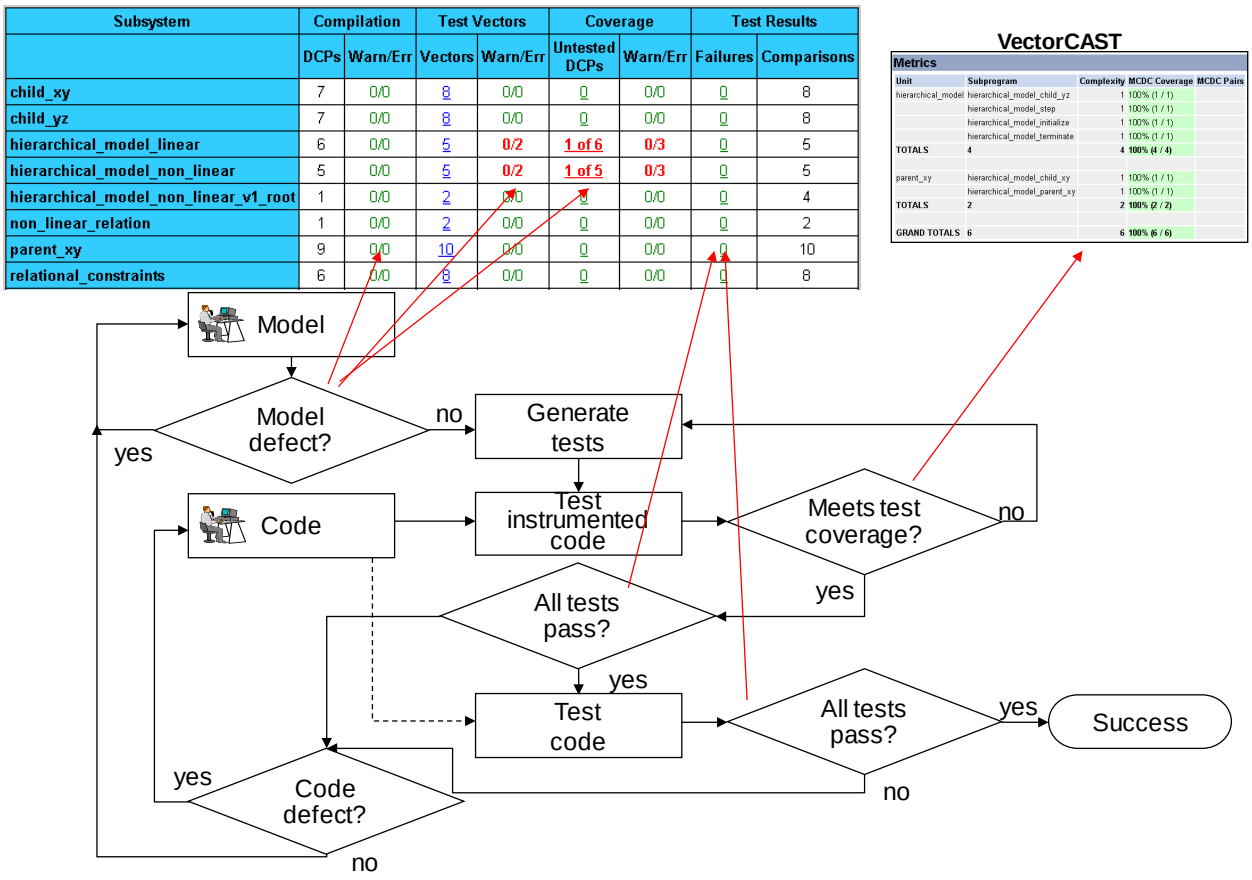


Figure 10. Model-Based Measures

3 Surveyed Technologies

This section discusses different tools and technologies in the classes identified in Section 2.

3.1 SMT

The SMT Competition of 2010 covered approximately 20 divisions, which relate to the different theories supported by the tools [6]. While there are different categories, some tools perform better over several of the theories, especially those including nonlinear and linear mixed and real arithmetic. Three tools used in experiments include:

- Yices – SRI, developers of PVS [14]
- CVC3 - originated at Stanford University with the SVC system, but is now being evolved by New York University and University of Iowa [15]
- Z3 – Microsoft [16]

Equally important is that when the models produce "SAT" they return what is referred to as a model, with values that satisfy the constraints. These values can support testing, but as shown in the experiment discussed in 4.1, the selected values are not necessarily at the boundaries or subdomain boundaries and this limits the fault finding effectiveness. Section 4.1 provides details about the experiment testing the fault finding effectiveness of yices, CVC3 and T-VEC VGS.

Some sample cases assess linear and nonlinear constraints solving of yices, CVC3 and Z3 is described in Section 4.2.

3.2 Prover Plugin used by Simulink/Stateflow Design Verifier

The Prover plugin is an SMT solver that is used by the Mathworks Design Verifier for Simulink/Stateflow. Rockwell performed some case studies, including one with the Prover plugin for Simulink, and the report states [17]:

> Because of its extensive use of floating point numbers and large state space, the Effector Blender (EB) logic cannot be verified using a BDD-based model checker such as NuSMV[18]. Instead, the EB was analyzed using the Prover SMT-solver[3] from Prover Technologies. Even with the additional capabilities of Prover, several new issues had to be addressed, the hardest being dealing with floating point numbers.
>
> While Prover has powerful decision procedures for linear arithmetic with real numbers and bit-level decision procedures for integers, it does not have decision procedures for floating point numbers. Translating the floating point numbers into real numbers was rejected since much of the arithmetic in the EB is inherently nonlinear. Also, the use of real numbers would mask floating point arithmetic errors such as overflow and underflow.

Three points are drawn from the Rockwell study:

- Floating point numbers, especially for realistic systems with large state spaces are likely to be difficult to verify using model checkers
- Floating point numbers with nonlinear constraints that usually exist in systems need by NASA and DARPA pose a challenge for SMT solvers as the real number solvers don't address nonlinearity or floating point numbers
- Floating point numbers in real environments where underflows and overflows can occur mean that domains and ranges are needed for input and output variables

A tool chain evaluation identified similar issues with the Prover plugin for Simulink Design Verifier (DV) that required nonlinear operations as discussed in Section 2.2.3. The tool chain evaluation involved several tools; one comparison assessed the DV and T-VEC VGS ability to identify satisfiability issues in models as well as test generation capabilities. The evaluation used the model shown in Figure 8 that had two seed defects, one for a linear constraint that was unsatisfiable and one for a non-linear constraint that was unsatisfiable. The Prover plugin did not identify the unsatisfiable linear constraint and indicated a number of errors related to the satisfiable of linear constraints as shown in Figure 11. T-VEC VGS was able to identify both seeded defects, as reflected in the status generated by VGS shown in Figure 10.

Figure 11 shows some highlights from the DV generated reports. DV generates one or more test objectives for each Simulink blocks in an attempt to achieve MC/DC test coverage of the model. DV produces Satisfied, Unsatisfied or Error for each block as shown for the hierarchical_model_linear/child_yz/Logical Operator block. If satisfiable, it produces a test case similar to SMT solvers. Row 6 indicates that an error exists, but VGS generated test vectors

[3] Prover SMT-solver is the component used by the Mathworks Design Verifier

corresponding to that block are shown in the lower right of Figure 11. NOTE: this information was shared with the Mathworks senior executives.

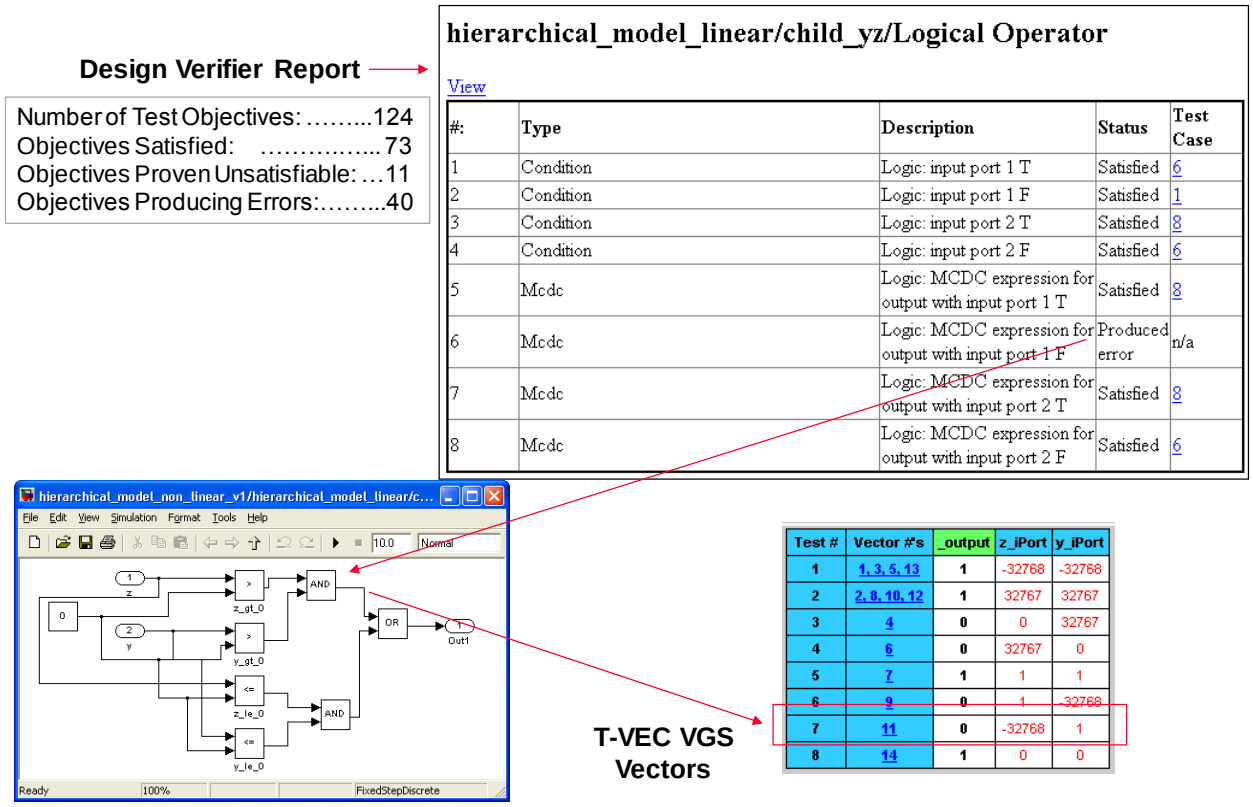
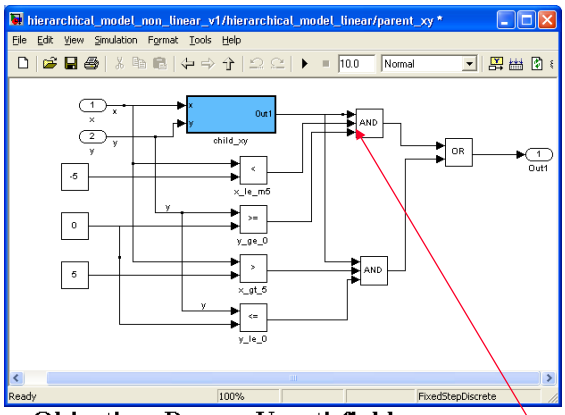


Figure 11. Design Verifier Results Compared to T-VEC VGS (1 of 2)

Figure 12 provides another report for those "Objective Proven Unsatisfiable." However, the vectors generated by T-VEC VGS shown in the upper right of Figure 12 identify several test vectors for some of the objectives classified by the DV as unsatisfiable.



Test #	Vector #'s	_output	x	y
1	1, 3, 7, 11, 13	0	(-32768)	(-32768)
2	2, 4, 6, 12, 16	0	(32767)	(32767)
3	5	0	-5	(-32768)
4	8	1	(32767)	-1
5	9	1	-32768	0
6	10	1	-6	32767
7	14	0	5	(32767)
8	15	1	(-32768)	1
9	17	1	6	-32768
10	18	1	32767	0

T-VEC VGS Vectors

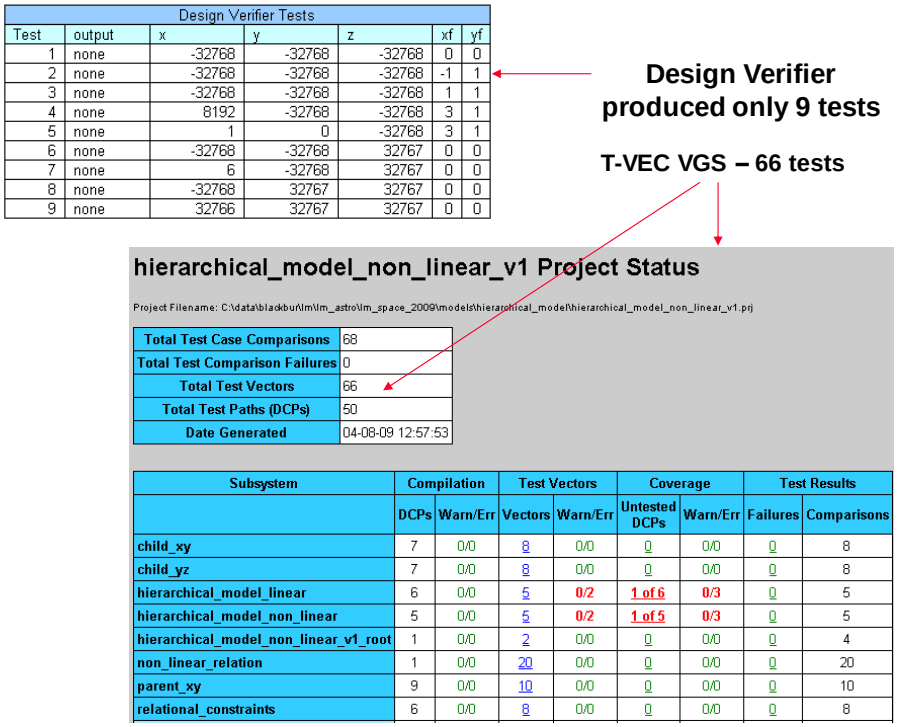
Objectives Proven Unsatisfiable

Simulink Design Verifier proved that there does not exist any test case exercising these test objectives. This often indicates the presence of dead-code in the model. Other p
model due to parameter configuration or test constraints such as given using Test Condition blocks. In rare cases, the approximations performed by Simulink Design Verifie

#:	Type	Model Item	Description	Test Case
54	Condition	hierarchical_model_linear/parent_xy/Logical Operator	Logic: input port 3 F	n/a
56	Mcdc	hierarchical_model_linear/parent_xy/Logical Operator	Logic: MCDC expression for output with input port 1 F	n/a
60	Mcdc	hierarchical_model_linear/parent_xy/Logical Operator	Logic: MCDC expression for output with input port 3 F	n/a
66	Condition	hierarchical_model_linear/parent_xy/Logical Operator4	Logic: input port 3 F	n/a
68	Mcdc	hierarchical_model_linear/parent_xy/Logical Operator4	Logic: MCDC expression for output with input port 1 F	n/a
72	Mcdc	hierarchical_model_linear/parent_xy/Logical Operator4	Logic: MCDC expression for output with input port 3 F	n/a
87	Condition	hierarchical_model_linear/Logical Operator	Logic: input port 4 T	n/a
89	Mcdc	hierarchical_model_linear/Logical Operator	Logic: MCDC expression for output with input port 1 T	n/a
91	Mcdc	hierarchical_model_linear/Logical Operator	Logic: MCDC expression for output with input port 2 T	n/a
93	Mcdc	hierarchical_model_linear/Logical Operator	Logic: MCDC expression for output with input port 3 T	n/a
95	Mcdc	hierarchical_model_linear/Logical Operator	Logic: MCDC expression for output with input port 4 T	n/a

Figure 12. Design Verifier Results Compared to T-VEC VGS (2 of 2)

As shown Figure 13, T-VEC VGS produced 66 test vectors using the test generation mode that produces the fewest vectors possible that are needed to provide coverage for the model paths. The DV produced only nine test cases; note DV produces only test inputs, and does not produce the expected outputs. The MATLAB simulator can be used to simulate the execution of the inputs to produce the outputs.



Design Verifier Tests

Test	output	x	y	z	xf	yf
1	none	-32768	-32768	-32768	0	0
2	none	-32768	-32768	-32768	-1	1
3	none	-32768	-32768	-32768	1	1
4	none	8192	-32768	-32768	3	1
5	none	1	0	-32768	3	1
6	none	-32768	-32768	32767	0	0
7	none	6	-32768	32767	0	0
8	none	-32768	32767	32767	0	0
9	none	32766	32767	32767	0	0

hierarchical_model_non_linear_v1 Project Status

Project Filename: C:\data\blackbur\lm\lm_astro\lm_space_2009\models\hierarchical_model\hierarchical_model_non_linear_v1.prj

Total Test Case Comparisons	68
Total Test Comparison Failures	0
Total Test Vectors	66
Total Test Paths (DCPs)	50
Date Generated	04-08-09 12:57:53

Subsystem	Compilation		Test Vectors		Coverage		Test Results	
	DCPs	Warn/Err	Vectors	Warn/Err	Untested DCPs	Warn/Err	Failures	Comparisons
child_xy	7	0/0	8	0/0	0	0/0	0	8
child_yz	7	0/0	8	0/0	0	0/0	0	8
hierarchical_model_linear	6	0/0	5	0/2	1 of 6	0/3	0	5
hierarchical_model_non_linear	5	0/0	5	0/2	1 of 5	0/3	0	5
hierarchical_model_non_linear_v1_root	1	0/0	2	0/0	0	0/0	0	4
non_linear_relation	1	0/0	20	0/0	0	0/0	0	20
parent_xy	9	0/0	10	0/0	0	0/0	0	10
relational_constraints	6	0/0	8	0/0	0	0/0	0	8

Figure 13. Test Comparison of T-VEC and Design Verifier

3.3 Nonlinear Constraint Solvers

iSAT is an SMT solver that supports nonlinear constraints using interval constraint propagation with Boolean combinations and some nonlinear arithmetic constraints involving transcendental functions [19]. However, the support for nonlinear constraint solving has limitations as evidenced by iSAT's inability to solve some of our evaluation problems described the CalCS paper, and shown in Figure 14 [20].

CalCS, developed at Berkeley, is not available for download, but a paper discusses its capabilities to solve Boolean combinations of nonlinear constraints that are convex. It applies convex programming. The following constraints (8), (9), and (10) are claimed to show issues with iSAT. A summary is provided in Section 4.2.

$$(x_1^2 + x_2^2 - 1 \leq 0) \wedge (x_1^2 + x_2^2 - 6x_1 + 5 < 0), \quad (8)$$

iSAT returns an interval that contains a spurious solution, while our convex sub-theory can rigorously deal with tight inequalities and correctly returns UNSAT (see (8) and Conj3

$$(x+y < a) \wedge (x-y < b) \wedge (2x > a+b) \wedge (a = 1) \wedge (b = 0.1), \quad (9)$$

$$(x \leq 10^9) \wedge (x + p > 10^9) \wedge (p = 10^{-8}). \quad (10)$$

Figure 14. Constraint Expression from CalCS Paper

RealPaver software is capable of modeling and solving nonlinear systems [21]. It implements a modeling language and some interval-based algorithms to process systems of nonlinear

constraints over the real numbers. While the RealPaver results seem promising, there have not been additional published results since 2004. Other related solvers that were not evaluated include: iCOs, perPlex, GlobSol, Lurupa [22].

3.4 Floating-point Nonlinear Constraint Solvers

An approach for search-based floating point constraint solving for symbolic execution is being implemented as a plugin for the Microsoft Pex Dynamic Symbolic Execution (DSE) testing tool [23]. It combines Search-Based Software Testing (SBST) and DSE. It attempts to support test generation for floating point computations in DSE. Pex is a program-based approach to testing, which generates test inputs for .NET code, based on DSE. Test inputs are generated for parameterized unit tests, or for arbitrary methods of the code under test [24]. There is a download available for Pex, but under the terms of the license, it could not be evaluated.

3.5 SMT versus LP Solver

Linear programming has roots with the Simplex solver. Yices and Z3, re-implement from scratch a Simplex solver in exact rational arithmetic. From an SMT solver perspective, soundness is a requirement. Scalable off-the-shelf Simplex implementations, (e.g., GNU Linear Programming Kit [Glpk] and CPLex), are inexact because they use floating-point instead of exact rational arithmetic. When the Simplex method is used as a decision procedure for linear arithmetic, this is an issue because even the slightest approximation can be responsible for an unsound result [25].

Besson points out a practical way to deal with the potential soundness issues related to nonlinear and floating-point constraint solving. From a proof point-of-view, the approach uses an oracle as a witness; this provides a second independent source to judge the validity of the potentially unsound result. While this approach is reasonable, it does require additional effort using an interactive theorem prover. However, to fully automate the verification of an implementation against a formal specification, there are some other possible scenarios that have been discussed with certification authorities that leverage a similar independence argument:

- For example, as reflected in Figure 10:
  - If a nonlinear, floating-point constraint solver produces test inputs and expected outputs, and
  - Those tests are executed against an implementation, and the implementation produces the same outputs within tolerance as the specification-based solver, and
  - MC/DC test coverage is provided by the generated tests against the code
- This would provide verification evidence that should be adequate to support certification, because there are multiple and independent sources that agree.
- The only remaining issues are the adequacy of the generated test cases at exposing faults, which is discussed in Section 4.1, and qualification of the tools, which is not discussed in this paper.

3.6 Model Checking

There are many (more than 30) model checkers that check properties of specifications and programs. We looked at BLAST and attempted to evaluate SAL-ATG (version 3.0), but could not evaluate, because the download did not include ATG.

However, based on the statements provided by Rockwell [17 that discussed the issues of using model checkers such as NuSMV on floating point numbers, and as discussed in Section 2.2, the

paper "*Testing with model checkers: A survey*," which provides an excellent summary of model checking capabilities, history, and strategies for applying model checking to testing, we repeat the point cited [8]:

- A pilot study was conducted to investigate the suitability of condition based coverage criteria. In this experiment, test suites were generated using different condition based coverage criteria for a close to production model of a flight guidance system from Rockwell Collins Inc. The fault detection ability of the different test suites was measured on mutant versions of the model. The experiment showed that a set of randomly generated test cases generated using the same effort were superior to all coverage based test suites [10].

We emphasize that additional heuristics are needed to guide the selection of test data when the constraints include linear and nonlinear constraints. Model checkers, like SMT solvers, are inadequate currently.

3.7 Model-based Test Generation

There are references that Spec Explorer 2010 integrates with Z3, but in reviewing the documentation for Spec Explorer, it is unclear how to use Z3 with Spec Explorer other than through the Z3 API, which is applicable to several languages. Spec Explorer explores the state space of a given model, coded in C# using slicing and represents the explored model as state sequences. The test generation is based off of the sequences and user-specified controls such as pairwise, interaction, isolated, seeded, etc. to select values associated with the parameters to generate test case values (inputs, no expected outputs). Therefore, while there is significant automation within the tools to determine the sequences, the specific values selected are largely controlled by the model and user-specific controls, and not derived from the constraints in a manner described in the experiment of Section 4.1.

3.8 Other Verification and Analysis Systems

Interactive theorem provers (ITP) such as Prototype Verification System (PVS) [26], Isabelle [27], and Coq [28] are arguably powerful theorem provers. However, we are interested in automated verification that can be performed by typical engineers, and because these types of tools require additional expertise and rely on humans to guide the search for proofs, we did not survey this class of theorem provers.

KeYmaera is a hybrid interactive and automatic theorem prover. It supports nonlinear constraint, but no test generation [29]. We were not able to evaluate KeYmaera, because we did not have or want to purchase solver plugins such as Mathematica [30].

Alloy analyzes specifications written in Alloy specification language [31]. It can generate instances of model invariants, simulate execution of operations defined as part of the model, and check user-specified properties of a model using a SAT-based model finder.

4 Experiments

To provide objective data about the effectiveness of these tools in supporting automated verification, we conducted a few experiments. The first class of experiments compares the fault-finding effectiveness of tests generated from the tools. The second class of experiments evaluates how well the tools solve combinations of linear, nonlinear and floating point constraints to determine if a model (specification) is SAT or UNSAT.

4.1 Effective Test Data Comparison

The experiment used a vertical tracking functional specification. This type of function is found in an airborne traffic and collision avoidance systems (TCAS). The specification did not have nonlinear constraints nor did it use floating point variables. The goal of the experiment is to analyze the fault finding effectiveness of models produced by SMT-type tools. One technique that has been used to measure the effectiveness of test cases is called mutation analysis. Mutation testing is a fault-based testing technique that has been effective in assessing the adequacy of a test set for a program [32], [33], and in particular a subset of selective mutants [34] 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.

The specification was created from a set of requirements similar to those described in Appendix B.2. It was first modeled in the T-VEC Tabular Modeler and translated into T-VEC VGS that produces test vectors (i.e., inputs and expected outputs). An equivalent model was created using the SMT solvers CVC3 and yices. Both tools have the capability to produce a model (i.e., COUNTERMODEL), which includes values that satisfy the constraints. These values are used as test inputs in the experiment. Details of the experiment and mutation testing are provided in Appendix A.

The first criterion for the experiment is that the generated test cases must pass all test cases for the base (non-mutated) program; all were successful as summarized in Figure 15. While the specification of CVC3 and yices seem equivalent, the semantics of the expression were interpreted differently. Details are provided in Appendix A. Figure 15 provides also a summary of the test effectiveness score of T-VEC VGS compared with CVC3 and yices.

- T-VEC VGS produced 28 test vectors killing all mutants. T-VEC produced more test cases than CVC3 or yices, because it includes several built-in test generation heuristics. For example, VGS has test select heuristics to select low-bound and high-bound values, and for the absolute value function, it selects values of the inputs from both the negative and positive domain.
- Two versions of CVC3 were created killing approximately 33% and 39% of the mutants respectively. CVC3 v1 had only five vectors. CVC3 did not produce a counter model for one of the required outputs for both v1 and v2. In addition, CVC3 v1did not produce models for several disjunctions contained within the specification, even though the “CONTINUE” command was issued, which we assumed would produce additional cases if they are satisfiable. The CVC3 v2 version expanded the disjunctions explicitly in a manner similar to the yices specification.
- Yices produced eight test cases killing 50% of the mutants, using a set of formulas that expanded the disjunctions explicitly similar to CVC v2.

Experiment Parameters	Version 1			
Lines of Code	49			
Decisions	9			
Test Selection Approach	VGS	CVC3 v1	CVC3 v2	yices
Version	3.6.0	2.2	2.2	1.0.28
Base program	Pass	Pass	Pass	Pass
Test Cases	28	5	7	8
Seeded faults	18	18	18	18
Exposed faults	18	6	7	9
Effectiveness Score	100.0%	33.3%	38.9%	50.0%

Figure 15. Test Effectiveness Summary of Results

Developing the specifications for this experiment helped expose an important challenge discussed more in Section 5.1. Modeling languages such as the BAE FCSL DSML, shown in Figure 3, allows user to construct specifications modularly. This is important for scalability, readability, maintainability, etc. However, to leverage the analysis and test generation capabilities of other tool chain elements, these models must be transformed. During the transformation process it is important to preserve the traceability from the model to the derived specification. This allows for maintaining relationships to derived attributes such as model-to-test traceability. Initial versions of the specification attempted to preserve the modularity, but for the CVC3 v2 and yices specifications, the model had to be expanded completely; any modularity in a specification is lost as the model is fully expanded at one level in a flattened model.

4.2 Nonlinear and Floating Point Constraint Comparisons

The previous experiment focused on producing tests from specifications that are effective at exposing faults in implementations associated with that specification. In order to generate tests, the specification, usually associated with the requirements or design, should be defects free. As discussed in the objectives of this paper in Section 1.2, software often involves nonlinear constraints with floating point numbers. This experiment assesses tool capabilities in linear and nonlinear constraint solving with the additional consideration of performing the constraint solving with floating point variables. Table 1 shows example problems used in experiments with several of the tools. With the exception of CalCS [20], which is not available for evaluation, the following example problems were applied to the tools. The problems included:

- Hierarchical model (linear) shown in Figure 8 (tested for both sat and unsat) – this version did not include nonlinear constraints nor did it use floating point variables
- Hierarchical model (nonlinear) shown in Figure 8 (tested for both sat and unsat) – this version does include nonlinear constraints, and uses floating point numbers only with T-VEC VGS, Z3, and iSAT
- Challenges problems from the CalCS paper; the constraints from the CalCS paper are shown in Figure 14

Table 1. Results of Nonlinear and Linear Constraint Problems

Tool	Example Problems									Comments
	Evaluation license	Paper Analysis Only	hierarical model (linear) - SAT	hierarical model (linear) - UNSAT	hierarchical model (non-linear) - SAT	hierarchical model (non-linear) - UNSAT	CalCS 8	CalCS9	CalCS10	
T-VEC VGS	x		10	10	10	10	8	8	10	1
Yices	x		10	8	u	u				2, 3
Z3	x		10	10	10	10				4
CVC3	x		10	10	8	8				5
CalCS		x					+	+	+	6
iSAT	x		10	u	10	10	?	?	?	7, 8
Prover (Design Verifier)	x		5	?	?	?				9

A score of 10 indicates that the expected results were obtained. When the score is lower than 10 or a "?" is used, the following numbered comments describe aspects of the experiment that resulted in that lower score:

1. T-VEC VGS was able to solve the CalCS constraints, but the specification did need to reduce the initial floating point domains. The CalCS10 has constraints ranging from 10e9 to 10e-8; this range is larger than can be represented by a 64 bit floating point number, which has approximately 16 digits of precision.[4]
2. Yices was able to produce UNSAT for the hierarchical linear problem, but the problem needed to be represented more in an inline fashion, which is why the score is an 8. When a named term was used in a hierarchical manner, yices incorrectly produced SAT with the incorrect values. When the specification is flatted, yices properly produced UNSAT, which is expected.
3. Yices returned UNKNOWN (designated by the "u") for hierarchical model nonlinear for both SAT and UNSAT cases; this response means that yices detected a formula that contains nonlinear constraint, which it cannot solve and therefore returned UNKNOWN
   - We did not attempt the CalCS problems with yices
4. We did not attempt the CalCS problems with Z3
5. CVC3 had a similar issue to yices, and the expressions when represented in a hierarchical fashion caused CVC3 to return the incorrect results. Once flatted, in a manner similar to yices, CVC3 did produce the expected results.

[4] http://en.wikipedia.org/wiki/Double_precision_floating-point_format

- We did not attempt the CalCS problems

6. CalCS – we accepted that the claims made in the paper were valid [20], but were unable to apply the same problems to the tool, because there is no downloadable tool available for evaluation.
7. iSAT returned UNKNOWN for the hierarchical model (linear), when it should return UNSAT
8. We did not test the constraints documented in the CalCS paper against iSAT, but assumed the results summarized in the CalCS paper are accurate
9. Prover Plugin and Design Verifier

    - The Prover plugin and Design Verifier were run on a Simulink model provided to a Systems and Software Consortium client, who produced the results with a licensed version of DV. However as summarized in Section 3.2, the results were extracted from generated reports that showed errors or objective proven unsatisfiable when in both cases T-VEC VGS produced vectors for those satisfiable constraints.

5 Future Ideas

This section provides a summary of some ideas that are needed to leverage capabilities through tool chains to address the spectrum of issues required to reduce the cost of verification, while meeting the needs for usability and scalability.

5.1 Module Composition and Transformation

The SMT and model checking languages are appropriate for their intended use (i.e., specification analysis, proof of properties), but modeling languages and evolutionary capabilities such as the FCSL tool chain shown in Figure 3 are important for usability, readability, reviewability, scalability, manageability, and traceability. This form of modeling is needed for scaling to larger systems with teams of developers. Similar closing statements were made by the authors of the "*Testing with model checkers: A survey* [8] -

> …we need improved modeling techniques that can leverage the power of the tools.

This means that model translation and transformation are critical. Similar issues are described for other domain specific languages used by NASA [35].

During the process of converting formal specification developed as TTM and Simulink models we identified some unexpected interpretation with the CVC3, and yices language constructs. CVC3 and yices provide a language construct called a lambda expression that allows terms to be modularized, but the results produced were not interpreted in the same way. The SMT competition benchmark specifications are focused on speed and the expressions are often flatted into long, often difficult-to-read expressions. When the expressions are flatted, the expected results are produce. However, DMSL or modeling languages are often high-level, with the objectives of precision as well as readability and reviewability. They support modularity, which is critical for scalability. In order to leverage the analysis and test generation capabilities of other tool chain elements, these models must be transformed. During the transformation process we want to preserve the traceability from the model to the derived specification. This allows for maintaining relationships to derived attributes such as model-to-test traceability.

5.2 Translation Ordering

As noted in Section 1.1, performance issues with the T-VEC VGS generation process due to model transformation from the FCSL tool chain shown in Figure 3. During the analysis for this report we identified how certain placement of some types of translated expressions could reduce the execution time more than 25 times (162 seconds down to 6 seconds). These types of situations can occur when there are one or more levels of model translation, which is the case in this example. Additional model transformation rules can be added to change the ordering of the clauses and improve the speed; while the TTM and VGS languages are inherently declarative, the order of the constraint expressions does impact analysis and test generation speed.

Sections 5.1 and 5.2 identify model transformation as an area of research that is needed to link DMSL to tool chain elements to support formal analysis and automated test generation.

5.3 Heuristics for Nonlinear and Floating Point

This paper has focused on nonlinear and floating point characteristics that have impact on achieving the DARPA META program goals. These two characteristics of software might have the most impact on the challenge to reduce the verification effort.

This research for this paper identified different approaches, such as convex hull used in CalCS, and interval contraction. While T-VEC VGS operates on an "interval," it uses different heuristics, referred to as convergence, to solve constraints and select test values at the boundaries or subdomain boundaries of the polyhedron associated with the constraints on the inputs. These heuristics are different from the FloPSy effort discussed in Section 3.4 [24]. There are opportunities to leverage the speed of decision procedures of SMT in parallel with alternative nonlinear and floating point constraint solving by creating a hybrid mechanism as discussed in Section 5.4.

5.4 Apply SMT in Parallel

To address the speed of vector generation, a hybrid approach could be implemented to speed up the proof of properties and vector generation time. Here's a scenario:

- VGS produces vectors on a hierarchy of Domain Convergence Path (DCP).
    - Translation of a requirement or design model results in a hierarchy of DCPs, as represented in Figure 16. 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. This level-by-level process provides an efficient means for performing unit, software-integration, and system-level testing, while allowing traceability back to the hierarchical modeled requirements

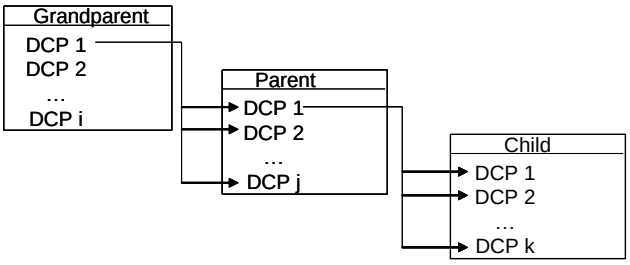


Figure 16. Hierarchical Subsystem Relationships

- A DCP is in the proper form for making a call to an SMT tool to determine satisfiability of the DCP represented as a formula
- An approach could be developed to send a DCP to a SMT solver in parallel; this should be easy to do as many processors today are multi-core. In parallel VGS processes the DCP using convergence, which is known to support nonlinear and floating point constraints. If the SMT returns UNSAT, this could be communicated to VGS to go to next DCP, otherwise if SAT or UNKOWN (i.e., the formula contains some combination of nonlinear or floating point expressions) and VGS should continue in an attempt to produce a test vector. This approach requires only that SMT solvers do not produce UNSAT incorrectly.

A more details architectural perspective of these concepts is needed, but beyond the scope of this paper.

5.5 Modular Generation of Global Invariant

The concept of global invariant generation could provide a set of global constraints to a theorem prover that could reduce the theorem proving and test vector generation time. VGS has an existing construct in its language to support this directly. The key need is in the generation of global invariants; researchers in the SMT community have developed mechanisms for global invariant generation. Assuming that moving from Domain Specific Modeling environments is a likely trend in the future, the key need is to apply global invariant generation in a modular way so that the global invariants are maintained within the subsystems (i.e., modular hierarchy) as reflected by Figure 16. This is important to maintain within the module only those constraints that are truly invariant. This means that model transformation is tied to this strategy.

5.6 Interface Modeling Tools with Open Source Model Checkers

While the model checkers were not necessarily highly effective at generating tests that were effective at finding faults, they do have capabilities needed for model analysis. As discussed in the NASA cases, the challenge is translating modeling languages into a form to leverage the model checking capabilities. Open source model checking tools have been used with the SCRtool from the Naval Research Laboratory. TTM has an import and export function for this language. This subject needs further analysis.

5.7 Benchmarks for Producing Verification Evidence

SMT benchmarks have helped the community progress, but as discussed in this paper there are other types of verification evidence that are needed. This paper produced a few benchmarks and identified others such as those provided by the CalCS paper.

6 Summary

The objectives discussed in this paper do not cover all the verification needs, yet being involved with FAA certification, FDA, DoD, and NASA safety efforts, the examples discussed in this paper reflect on the types of verification evidence that is needed for certification efforts that can be 88% of the total development cost.

The adoption of formal methods will require practical modeling languages, such as the FCSL language and Simulink that can scale to large systems. The manual translation of modeling notation such as the SCR method and Simulink into several SMT tools has highlighted some challenges. We need to address model transformation, and potentially standards to better leverage tool chains that can produce the four types of verification evidence required for certifications: model defects, related to requirement and design issues, proof of properties, test generation for verification of the target, and test coverage to ensure completeness of the target-based verification evidence.

7 References

[1] Brat, Guillaume, V & V of Flight-Critical Systems, Safe & Secure Systems & Software Symposium, June 2010.

[2] Blackburn, Mark, Model-Driven Verification and Validation, Safe & Secure Systems & Software Symposium, June, 15-17 2010.

[3] MAKING FORMAL METHODS PRACTICAL, Marc Zimmerman, Mario Rodriguez, Benjamin Ingram, Masafumi Katahira, Maxime de Villepin, Nancy Leveson, MIT, Cambridge, MA

[4] Producible Adaptive Model-based Software (PAMS) technology to the development of safety critical flight control software. PAMS has been developed under the Defense Advanced Research Projects Agency (DARPA) Disruptive Manufacturing Technologies program. Contract # N00178-07-C-2011.

[5] MODEL-BASED ADAPTATION OF FLIGHT-CRITICAL SYSTEMS, Sumit Ray, BAE Systems, Johnson City, New York, Gabor Karsai, Vanderbilt University, Nashville, Tennessee , Kevin M. McNeill, BAE Systems, Arlington, Virginia, Digital Avionics Systems Conference, 2009.

[6] Heitmeyer, Constance, Alan Bull, Carolyn Gasarch, Bruce Labaw, June 1995, SCR: A Toolset for Specifying and Analyzing Requirements, Gaithersburg, MA, Tenth International Conference on Computer Assurance, pp. 109-122.

[7] Barrett, C., A. Oliveras, M. Deters, A. Stump, Satisfiability Modulo Theories Competition (SMT-COMP) 2010: Rules and Procedures, http://www.smtcomp.org, 2010.

[8] Testing with model checkers: a survey, Gordon Fraser, Franz Wotawa, Paul E. Ammann, SNA-TR-2007-P2-04.

[9] Testing with model checkers: a survey, Gordon Fraser, Franz Wotawa, Paul E. Ammann, Software Testing, Verification and Reliability, Volume 19, Issue 3, pages 215–261, September 2009.

[10] Mats Per Erik Heimdahl, George Devaraj, and Robert Weber. Specification Test Coverage Adequacy Criteria = Specification Test Generation Inadequacy Criteria? In HASE, pages 178–186. IEEE Computer Society, 2004. ISBN 0-7695-2094-4.

[11] Automated Test Generation with SAL, Gregoire Hamon, Leonardo de Moura and John Rushby, 2005.

[12] White, L.J., and E.I. Cohen, A Domain Strategy for Computer Program Testing, IEEE Transactions on Software Engineering, 17, 7, July, 1980.

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

[14] Dutertre, B., L. de Moura, The YICES SMT Solver, Computer Science Laboratory, SRI International, http://yices.csl.sri.com/tool-paper.pdf.

[15] CVC3, http://cs.nyu.edu/acsys/cvc3/.

[16] Z3, http://research.microsoft.com/en-us/um/redmond/projects/z3/.

[17] Software Model Checking Takes Off, Miller, Whalen, Cofer, Feb. 2010.

[18] NuSMV http://nusmv.fbk.eu/.

[19] iSAT, http://gforge.avacs.org/.

[20] Nuzzo, P. A. Puggelli, S. Seshia, A. Sangiovanni-Vincentelli, CalCS: SMT Solving for Non-Linear Convex Constraints, Formal Methods in Computer Aided Design, October, 2010.

[21] RealPaver: An Interval Solver using Constraint Satisfaction Techniques, Granvilliers, Benhamou, 2006.

[22] Verified Linear Programming – a Comparison, Keil, 2006. http://www.ti3.tu-harburg.de/~keil/pub/VLPaC-S.pdf.

[23] Pex, http://research.microsoft.com/en-us/projects/pex/

[24] FloPSy - Search-Based Floating Point Constraint Solving for Symbolic Execution, Lakhotia, Tillmann, Harman, de Halleux, 2010.

[25] Besson, F., On using an inexact floating-point LP solver for deciding linear arithmetic in an SMT solver, SMT Workshop 2010, http://www.irisa.fr/celtique/fbesson/floating_point_simplex.pdf.

[26] PVS, http://www.csl.sri.com/projects/pvs/.

[27] Isabelle, http://www.cl.cam.ac.uk/research/hvg/isabelle/.

[28] Coq, http://coq.inria.fr/.

[29] KeYmaera, http://symbolaris.com/info/KeYmaera.html.

[30] Mathematica, http://www.wolfram.com/mathematica/.

[31] Alloy, http://alloy.mit.edu/community/.

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

[33] 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.

[34] Offutt, J., G. Rothermel, C. Zapf, An Experimental Evaluation of Selective Mutation, Fifteenth International Conference on Software Engineering, May 1993.

[35] Simmons, R., C. Pecheur, G. Srinivasan, Towards Automatic Verification of Autonomous Systems, IEEE/RSJ International conference on Intelligent Robots & Systems, 2000.

A. Appendix – Fault Finding Effectiveness Information

This section describes an experiment to compare the fault-finding effectiveness of test generated from T-VEC VGS and the model outputs from CVC3 and yices. The experiment uses a fault-seeding technique, referred to as mutation testing. 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 a subset of selective mutants.[5] 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.

Figure 17 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 TTM, CVC3 and yices. Test vectors were generated from the TTM model using T-VEC VGS. Test cases are extracted from the countermodel produced by CVC3 and yices. The base program executes correctly with respect to the test cases produced from the specification for VGS, CVC3, and yices. For this experiment, **correct** means that the: the actual output values must equal the expected output values for all test inputs.

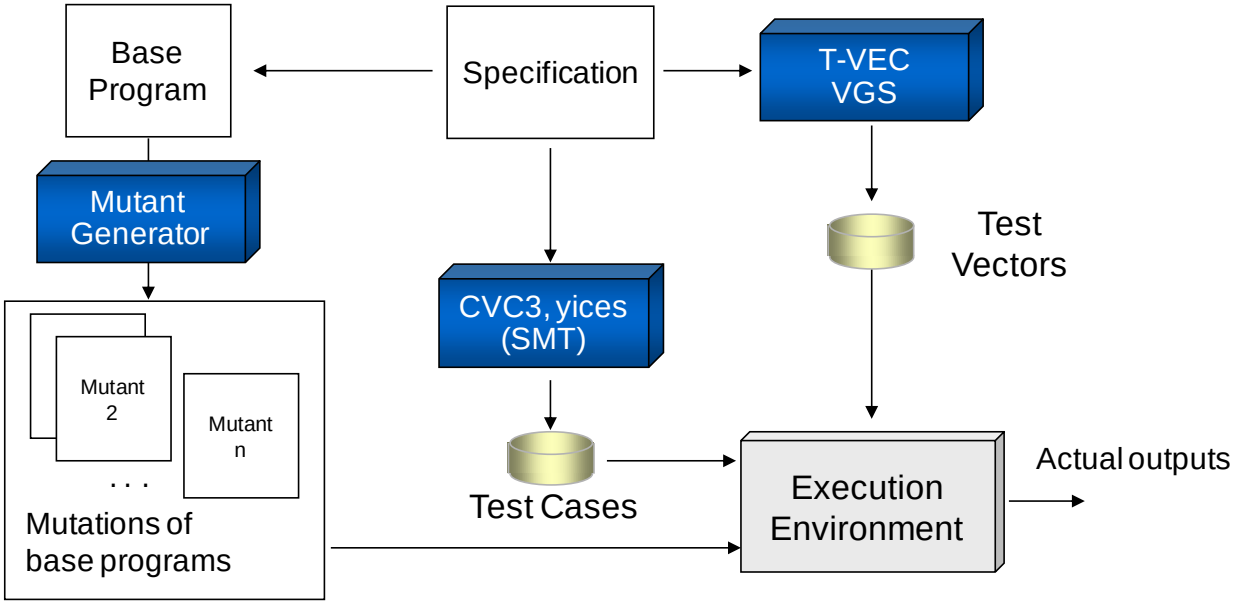


Figure 17. Experimental Design

The base program was mutated using the selective mutants. The selective mutant operators summarized in Table 2 were applied to those constructs used in the base program. The mutations resulted in 18 mutants. The test sets, both generated from VGS, CVC3 and yices, were executed by each mutant, and the actual output was recorded. If the actual output equaled the expected output, the test cases did not expose the mutant (aka "kill" the mutant).

Table 2. Select Mutant Operators

[5] 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.

Function	Was	Mutant
Relational operator replacement	<=	<
	<	>
	>	>=
	>=	>
	==	!=
Arithmetic operator replacement	+	-
	-	+
	/	-
Logical connector replacement	and	or
	or	and

A.1 TTM Models

The TTM modeling notation is derived the Software Cost Reduction method.

Behavior: trackingState (of type: trackingStateStrType)

#	Assignment	Condition	Mode
1	state = FAIL; pred_alt = OUTSIDE_WINDOW;	TRUE	NOT_TRACKING
2	state = FAIL; pred_alt = OUTSIDE_WINDOW;	currentStatus = BAD AND previousStatus = BAD	TRACKING
3	state = INIT; pred_alt = ownAltitude - nearestAltitude;	currentStatus = GOOD AND previousStatus = BAD AND t_inAltitudeWindow AND t_altRate <= 100	TRACKING
4	state = INIT; pred_alt = OUTSIDE_WINDOW;	currentStatus = GOOD AND previousStatus = BAD AND NOT (t_inAltitudeWindow)	TRACKING
5	state = COAST; pred_alt = lastAltitude;	currentStatus = BAD AND previousStatus = GOOD AND t_inAltitudeWindow	TRACKING
6	state = NOT_IN_VOLUME; pred_alt = OUTSIDE_WINDOW;	currentStatus = GOOD AND previousStatus = GOOD AND NOT (t_inAltitudeWindow)	TRACKING
7	state = IN_TRACK; pred_alt = (t_altRate / 2.0 + t_altRate);	currentStatus = GOOD AND previousStatus = GOOD AND t_inAltitudeWindow AND t_altRate <= 100	TRACKING
8	state = TOO_FAST; pred_alt = OUTSIDE_WINDOW;	currentStatus = GOOD AND previousStatus = GOOD AND t_inAltitudeWindow AND t_altRate > 100	TRACKING

Behavior: t_altRate (of type: altitudeType)

#	Assignment	Condition
1	abs(lastAltitude - ownAltitude)	TRUE

Behavior: t_inAltitudeWindow (of type: Boolean)

#	Assignment	Condition
1	TRUE	abs(ownAltitude - nearestAltitude) <= 2700
2	FALSE	abs(ownAltitude - nearestAltitude) > 2700

A.1 CVC3 Version 1

The following is the CVC3 specification created that is intended to be equivalent to the TTM model shown above. See the http://www.cs.nyu.edu/acsys/cvc3/doc/user_doc.html for a description of the language.

```
%%% Description:  Vertical tracker for CVC3 (Version 1)
%%%               This is an provides a specification for a vertical tracker function.
%%%               This should be equivalent to the one produced in TTM.
%%% Create by:    Mark Blackburn
%%% Last update:  13-Nov-2010

altitudeType: TYPE = SUBTYPE(LAMBDA (x: INT): x >= -1000 AND x <= 127000, 0);
DATATYPE
  statusType =  BAD | GOOD
END;
DATATYPE
```

```
  trackingStateType = FAIL | INIT | COAST | IN_TRACK | NOT_IN_VOLUME | TOO_FAST | NONE
END;

output: trackingStateType;
currentStatus, previousStatus : statusType;
lastAltitude, nearestAltitude, ownAltitude, pred_alt: altitudeType;

OUTSIDE_WINDOW: altitudeType = 2701;

t_altRate: altitudeType =
  IF lastAltitude >= ownAltitude THEN lastAltitude - ownAltitude
  ELSE ownAltitude - lastAltitude ENDIF;

AltitudeWindow: altitudeType =
  IF nearestAltitude > ownAltitude THEN nearestAltitude - ownAltitude
  ELSE ownAltitude - nearestAltitude ENDIF;

t_inAltitudeWindow: BOOLEAN = AltitudeWindow >= -2700 AND AltitudeWindow <= 2700;

m_tracking: BOOLEAN = ownAltitude >= 10000;

trackingState: trackingStateType =
  IF (NOT m_tracking OR m_tracking AND currentStatus = BAD AND previousStatus = BAD)
         AND pred_alt = OUTSIDE_WINDOW
         AND output = FAIL
    THEN FAIL
  ELSIF m_tracking
       AND currentStatus = GOOD
       AND previousStatus = GOOD
       AND NOT t_inAltitudeWindow
       AND pred_alt = OUTSIDE_WINDOW
       AND output = NOT_IN_VOLUME
    THEN NOT_IN_VOLUME
  ELSIF m_tracking
       AND currentStatus = GOOD
       AND previousStatus = BAD
       AND ((NOT t_inAltitudeWindow
           AND pred_alt = OUTSIDE_WINDOW)
           OR
           (t_inAltitudeWindow
           AND pred_alt = ownAltitude - nearestAltitude))
       AND output = INIT
    THEN INIT
  ELSIF m_tracking
       AND currentStatus = BAD
       AND previousStatus = GOOD
       AND t_inAltitudeWindow
       AND pred_alt = lastAltitude
       AND output = COAST
    THEN COAST
  ELSIF m_tracking
       AND currentStatus = GOOD
       AND previousStatus = GOOD
       AND t_inAltitudeWindow
       AND t_altRate <= 100
       AND pred_alt = (t_altRate / 2 + t_altRate)
       AND output = IN_TRACK
     THEN IN_TRACK
  ELSIF m_tracking
       AND currentStatus = GOOD
       AND previousStatus = GOOD
       AND t_inAltitudeWindow
       AND t_altRate > 100
       AND pred_alt = OUTSIDE_WINDOW
       AND output = TOO_FAST
    THEN TOO_FAST
  ELSE
    NONE
  ENDIF;

PUSH;
```

```
CHECKSAT trackingState=FAIL;
COUNTERMODEL;
CONTINUE;
POP;

PUSH;
CHECKSAT trackingState=INIT;
COUNTERMODEL;
CONTINUE;
POP;

PUSH;
CHECKSAT trackingState=COAST;
COUNTERMODEL;
POP;

PUSH;
CHECKSAT trackingState=TOO_FAST;
COUNTERMODEL;
POP;

PUSH;
CHECKSAT  trackingState=NOT_IN_VOLUME;
COUNTERMODEL;
POP;

PUSH;
CHECKSAT trackingState=IN_TRACK;
COUNTERMODEL;
POP;
```

A.1 CVC3 Version 2

```
%%% Description:  Vertical tracker for CVC3 (Version 2)
%%%               This is an provides a specification for a vertical tracker function.
%%%               This should be equivalent to the one produced in TTM.
%%% Create by:    Mark Blackburn
%%% Last update:  14-Nov-2010

altitudeType: TYPE = SUBTYPE(LAMBDA (x: INT): x >= -1000 AND x <= 127000, 0);
DATATYPE
  statusType =  BAD | GOOD
END;
DATATYPE
  trackingStateType = FAIL | INIT | COAST | IN_TRACK | NOT_IN_VOLUME | TOO_FAST | NONE
END;

output: trackingStateType;
currentStatus, previousStatus : statusType;
lastAltitude, nearestAltitude, ownAltitude, pred_alt: altitudeType;

OUTSIDE_WINDOW: altitudeType = 2701;

t_altRate: altitudeType =
  IF lastAltitude >= ownAltitude THEN lastAltitude - ownAltitude
  ELSE ownAltitude - lastAltitude ENDIF;

AltitudeWindow: altitudeType =
  IF nearestAltitude > ownAltitude THEN nearestAltitude - ownAltitude
  ELSE ownAltitude - nearestAltitude ENDIF;

t_inAltitudeWindow: BOOLEAN = AltitudeWindow >= -2700 AND AltitudeWindow <= 2700;

m_tracking: BOOLEAN = ownAltitude >= 10000;

PUSH;
ASSERT NOT m_tracking AND pred_alt = OUTSIDE_WINDOW AND output = FAIL;
CHECKSAT;
COUNTERMODEL;
CONTINUE;
POP;
```

```
PUSH;
ASSERT m_tracking AND currentStatus = BAD AND previousStatus = BAD
         AND pred_alt = OUTSIDE_WINDOW
         AND output = FAIL;
CHECKSAT;
COUNTERMODEL;
POP;

PUSH;
ASSERT m_tracking AND currentStatus = GOOD AND previousStatus = BAD
       AND NOT t_inAltitudeWindow AND pred_alt = OUTSIDE_WINDOW AND output = INIT;
CHECKSAT;
COUNTERMODEL;
POP;

PUSH;
ASSERT m_tracking AND currentStatus = GOOD AND previousStatus = BAD
       AND t_inAltitudeWindow AND pred_alt = ownAltitude - nearestAltitude
       AND output = INIT;
CHECKSAT;
COUNTERMODEL;
POP;

PUSH;
ASSERT m_tracking AND currentStatus = BAD AND previousStatus = GOOD
       AND t_inAltitudeWindow AND pred_alt = lastAltitude AND output = COAST;
CHECKSAT;
COUNTERMODEL;
POP;

PUSH;
ASSERT m_tracking AND currentStatus = GOOD AND previousStatus = GOOD
       AND t_inAltitudeWindow AND t_altRate > 100 AND pred_alt = OUTSIDE_WINDOW
       AND output = TOO_FAST;
CHECKSAT;
COUNTERMODEL;
POP;

PUSH;
ASSERT m_tracking AND currentStatus = GOOD AND previousStatus = GOOD
       AND NOT t_inAltitudeWindow AND pred_alt = OUTSIDE_WINDOW AND output = NOT_IN_VOLUME;
CHECKSAT;
COUNTERMODEL;
POP;

PUSH;
ASSERT m_tracking AND currentStatus = GOOD AND previousStatus = GOOD AND t_inAltitudeWindow
       AND t_altRate <= 100 AND pred_alt = (t_altRate / 2 + t_altRate)
       AND output = IN_TRACK;
CHECKSAT;
COUNTERMODEL;
POP;
```

A.2 Yices

```
;;; Description:   Vertical tracker for yices.
;;;                This is an provides a specification for a vertical tracker function.
;;;                This should be equivalent to the one produced in TTM.
;;;  Create by:    Mark Blackburn
;;;  Last update:  10-Nov-2010
(set-evidence! true)
(set-verbosity! 3)
(define-type altitudeType(subtype (v::int) (and (>= v -1000) (<= v 127000))))
(define-type statusType (scalar BAD GOOD))
(define-type trackingStateType (scalar FAIL INIT COAST IN_TRACK NOT_IN_VOLUME TOO_FAST
NONE))

(define output:: trackingStateType)
(define currentStatus:: statusType)
```

```
(define previousStatus:: statusType)
(define lastAltitude:: altitudeType)
(define nearestAltitude:: altitudeType)
(define ownAltitude:: altitudeType)
(define pred_alt:: altitudeType)
(define OUTSIDE_WINDOW::altitudeType)

(define t_altRate::altitudeType (if (>= lastAltitude ownAltitude) (- lastAltitude
ownAltitude) (- ownAltitude lastAltitude)))

(define AltitudeWindow::altitudeType (if (> nearestAltitude ownAltitude) (- nearestAltitude
ownAltitude) (- ownAltitude nearestAltitude )))

(define t_inAltitudeWindow::bool (and (>= AltitudeWindow -2700) (<= AltitudeWindow 2700)))

(define m_tracking::bool (>= ownAltitude 10000))

(assert (= OUTSIDE_WINDOW 2701))

(push)
(echo "FAIL\n")
(assert (and (not m_tracking) (= pred_alt OUTSIDE_WINDOW) (= output FAIL)))
(check)
(pop)

(push)
(echo "FAIL\n")
(assert (and (= m_tracking true) (= currentStatus BAD)(= previousStatus BAD)(= pred_alt
OUTSIDE_WINDOW) (= output FAIL)))
(check)
(pop)

(push)
(echo "INIT\n")
(assert (and (= m_tracking true) (= currentStatus GOOD)(= previousStatus BAD) (not
t_inAltitudeWindow) (= pred_alt OUTSIDE_WINDOW) (= output INIT)))
(check)
(pop)

(push)
(echo "INIT\n")
(assert (and (= m_tracking true) (= currentStatus GOOD)(= previousStatus BAD) (=
t_inAltitudeWindow true) (= pred_alt (- ownAltitude nearestAltitude)) (= output INIT)))
(check)
(pop)

(push)
(echo "NOT_IN_VOLUME\n")
(assert (and (= m_tracking true) (= currentStatus GOOD)(= previousStatus GOOD) (not
t_inAltitudeWindow) (= pred_alt OUTSIDE_WINDOW) (= output NOT_IN_VOLUME)))
(check)
(pop)

(push)
(echo "COAST\n")
(assert (and (= m_tracking true) (= currentStatus BAD)(= previousStatus GOOD) (=
t_inAltitudeWindow true) (= pred_alt lastAltitude) (= output COAST)))
(check)
(pop)

(push)
(echo "IN_TRACK\n")
(assert (and (= m_tracking true) (= currentStatus GOOD)(= previousStatus GOOD) (=
t_inAltitudeWindow true) (<= t_altRate 100)
             (= pred_alt (+ t_altRate (/ t_altRate 2)))
             (= output IN_TRACK)))
(check)
(pop)

(push)
(echo "TOO_FAST\n")
```

```
(assert (and (= m_tracking true) (= currentStatus GOOD)(= previousStatus GOOD) (=
t_inAltitudeWindow true) (> t_altRate 100)
             (= pred_alt OUTSIDE_WINDOW)
             (= output TOO_FAST)))
(check)
(pop)
```

B. 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.[6]

B.1 Testing Strategies

Testing strategies are typically characterized as structural, that is, related directly to an implementation, and functional, that is, based on a specification.[7] 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. [8]

The following three types of errors can be made in an implementation, resulting in a fault:[9]

- 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.

---

[6] 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

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

[8] 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.

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

- 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 and Cohen proposed ***domain testing theory*** as a strategy for selecting test points to reveal domain errors.[10] 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.[11]

Beizer 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. [12] 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.

B.2 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 18 provides some additional details about the Vertical Tracker. The altitude of an aircraft 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 18, 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.

---

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

[11] 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.

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

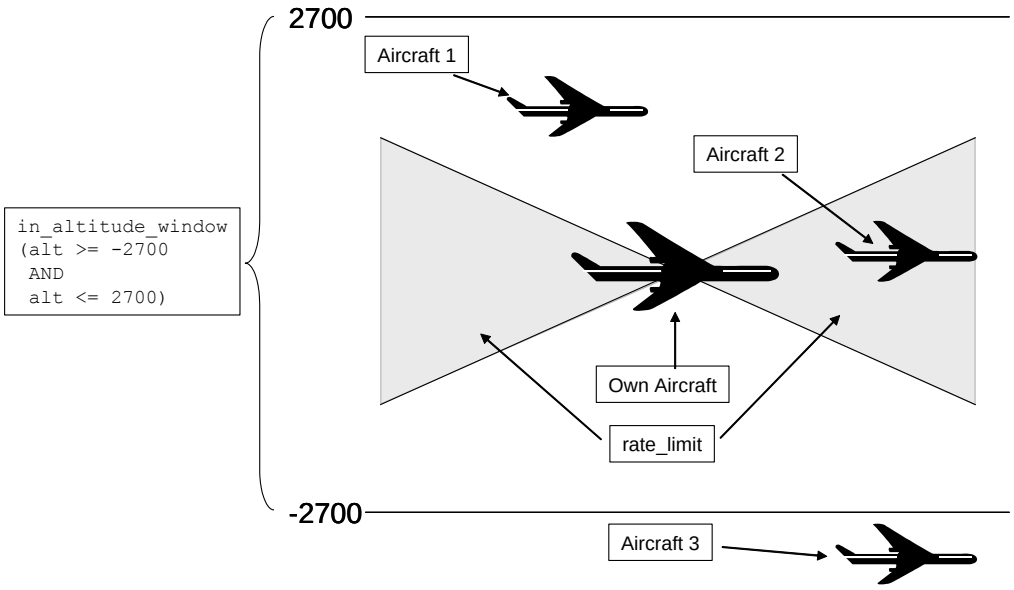


Figure 18. Visual Representation of Vertical Tracker Constraints

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 19. For this example, Aircraft 2 is the Nearest_altitude, as shown in Figure 18.

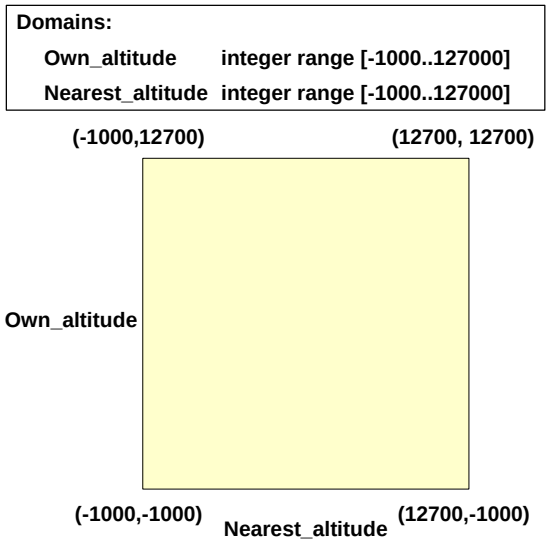


Figure 19. Input Domains

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 20, 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).

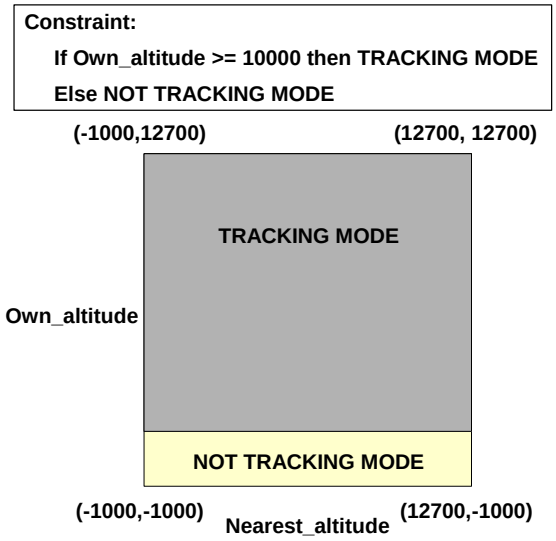


Figure 20. Partition of Domain Into Subdomains

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 21. Figure 22 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.

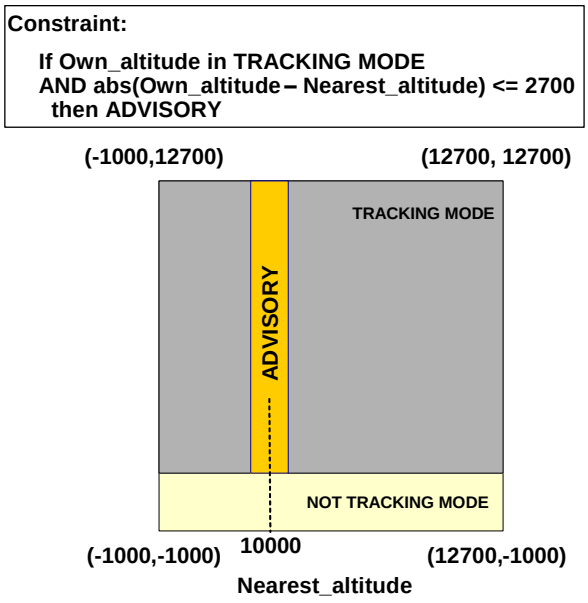
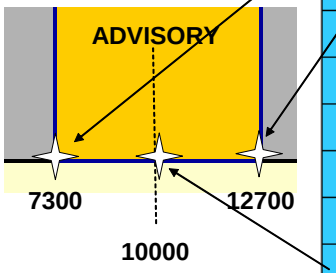


Figure 21. Constraints for Advisory Mode



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 22. Sample Test Points

B.3 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 21, 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 called 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 22. 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 3 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 3. 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

B.4 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.

B.5 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 23:

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
```

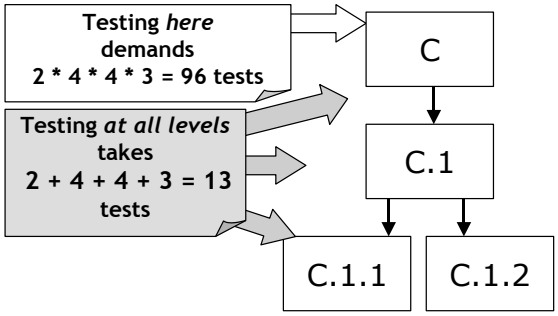


Figure 23. Example Illustrating Paths and Test Coverage

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.

T-VEC VGS supports hierarchical relationships. In generating test vectors for a hierarchy of models, as represented in Figure 24, 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 23. This level-by-level process provides an efficient means for performing unit, software-integration, and system-level testing.

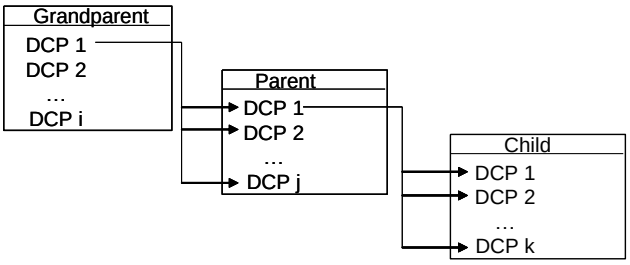


Figure 24. Hierarchical Subsystem Relationships

C. SMT Theories

The following list the different divisions of the SMT competition. QF_ stands for quantifier free.

- QF UF: uninterpreted functions.
- QF RDL: real difference logic.
- QF IDL: integer difference logic.
- QF UFIDL: uninterpreted functions and integer difference logic.

- QF UFLIA: uninterpreted functions and linear integer arithmetic.
- QF UFLRA: uninterpreted functions and linear real arithmetic.
- QF UFNRA: uninterpreted functions and nonlinear real arithmetic.
- QF LRA: linear real arithmetic.
- QF LIA: linear integer arithmetic.
- QF NIA: nonlinear integer arithmetic.
- QF AX: arrays with extensionality.
- QF AUFLIA: arrays, uninterpreted functions and linear integer arithmetic.
- QF BV: fixed-width bitvectors.
- QF AUFBV: arrays, fixed-width bitvectors and uninterpreted functions.
- LRA: (quantified) linear real arithmetic.
- AUFLIA+p: (quantified) arrays, uninterpreted functions and linear integer arithmetic, patterns included.
- AUFLIA−p: (quantified) arrays, uninterpreted functions and linear integer arithmetic, patterns not included.
- AUFLIRA: (quantified) arrays, uninterpreted functions and mixed linear integer and real arithmetic.
- UFNIA+p: (quantified) uninterpreted functions and nonlinear integer arithmetic, patterns included.
- AUFNIRA: (quantified) arrays, uninterpreted functions and mixed nonlinear integer and real arithmetic.

D. Trademarks