# USING MODELS FOR TEST GENERATION AND ANALYSIS

*Mark R. Blackburn, Software Productivity Consortium, Herndon, Virginia*

## Introduction

Systems are increasing in complexity. More systems perform mission-critical functions, and dependability requirements such as safety, reliability, availability, and security are vital to the users of these systems. The competitive marketplace is forcing companies to define or adopt new approaches to reduce the time-to-market as well as the development cost of these critical systems. Much focus has been placed on front-end development efforts, not realizing that testing accounts for 40 to 75 percent of the lifetime development and maintenance costs [Bei83; GW94]. Testing is traditionally performed at the end of development, but market-driven schedules often force organizations to release products before they are adequately tested. The long-term effect is increased warranty costs due to products' poor reliability and poor quality.

Model-based development tools are increasing in use because they provide tangible benefits by supporting simulation and code generation, in addition to the traditional design and analysis activities. These tools help users develop requirement and design models of target systems. Certain tools are based on formal models, and the underlying models are represented using specification languages. Such formal specifications provide a basis for test case generation. However, the underlying development models are generally not represented in a form that supports automatic test case generation. The key challenge is to translate development-oriented modeling languages into a form that is suitable for automated test vector generation, specification-based test coverage analysis, requirement-to-test traceability, and design-to-test traceability.

### *Using Models for Testing and Analysis*

Figure 1 illustrates a conceptual view for using models to support test generation and analysis. Models and their associated tools typically provide various views of the system under development. When modeling tools are based on precise semantics, user models can also support:

- ***Test Vector Generation***. A test vector includes inputs, expected outputs, and an association with the specification from which it was derived
- ***Static Analysis***. Typically used to determine if there are contradictions in the specification
- ***Dynamic Analysis***. Analysis based on execution of the model.

Modeling tools are beginning to support simulation and code generation. Simulation of a model can help developers assess the correctness of the model with respect to the user requirements; however, it can be time consuming to develop simulation data required for thorough dynamic analysis. Automatically generated test vectors can provide a cost-effective way to exercise a model in a simulator using the boundary values associated with the constraints of a model specification; it is at the boundaries where model anomalies are typically discovered. In addition, these same test vectors can also be used to test the code in a host or target environment.

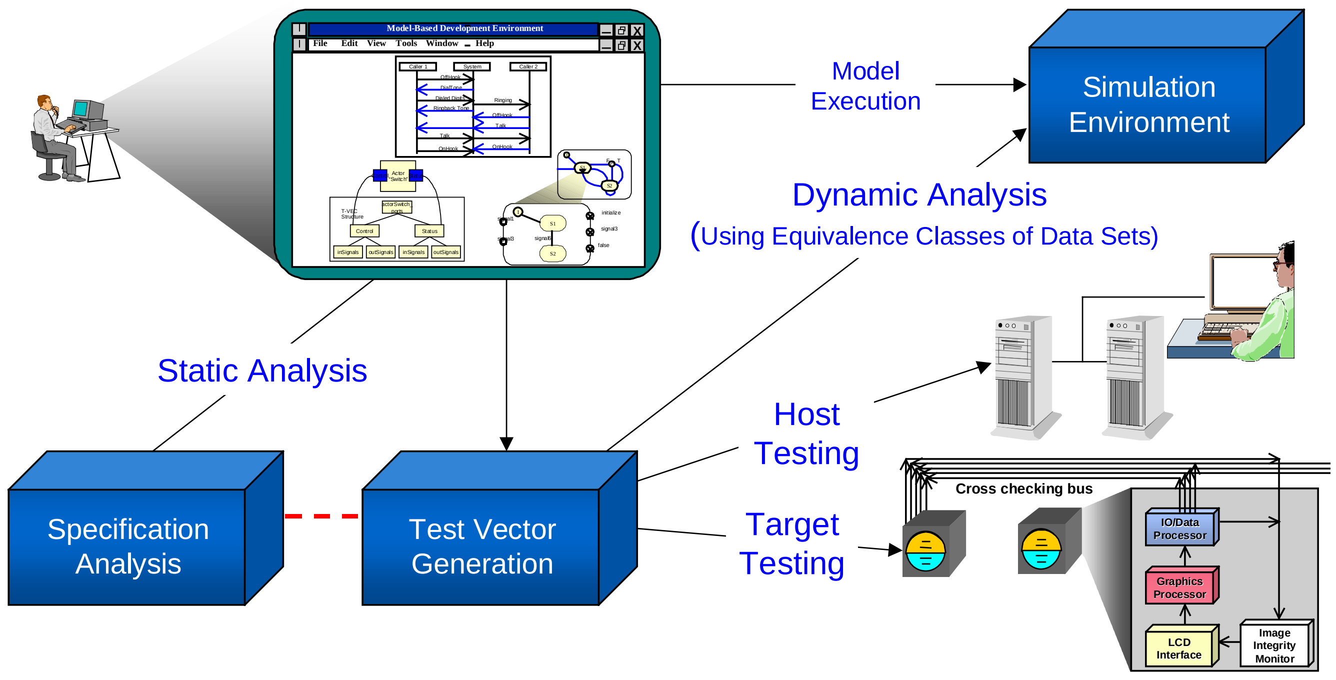


**Figure 1. Using Models for Test Generation and Analysis**

***Scope***

This paper describes the use of automated test generation and analysis from specification models. Through the integration of commercial off-the-shelf (COTS) model-development and test generation tools, a process has been developed that eliminates most of the traditional testing activities. This approach has been demonstrated to identify many types of specification errors prior to any implementation. This paper is based on the experiences in developing two model translators [BF98; Bla98] supporting:

- Software Cost Reduction (SCR) [Hen80]/Consortium Requirements Engineering Method (CoRE) [SPC93] for modeling requirements
- Real-Time Object-Oriented Modeling (ROOM) [SGW94] method for analysis and design

For each respective method and associated tool, the translators produce a specification that is used by the T-VEC tool system to generate test vectors and perform specification-based test coverage analysis. The model transformation process is briefly described using a specification example. The paper summarizes the results of applying the process and tools to industrial applications.

## Models and Specifications

Formal specifications provide simple abstract descriptions of the required behaviors describing what the software should do. Because formal specifications have, in the past, been considered difficult to use, they have not been widely used. Recent advances in visual model-based development tools provide the basis for developing formal specifications while hiding the formalism.

It has been commonly accepted that formal specifications provide a basis for test case generation. Goodenough and Gerhart may have been the first to claim that testing based only on a program implementation is fundamentally flawed [GG75]. Gourlay developed a mathematical framework for specification-based testing [Gou83]. Figure 2 graphically represents Gourlay's mathematical framework for testing and the key relationships between specifications, tests, and programs. Given a **specification** that describes the requirements for some system, there is one or more **programs** that implement the specification. **Tests** are derived from the specification; if every test executed by a program computes the appropriate expected results (i.e., passes every test), there is some level of confidence that the program satisfies the specification.

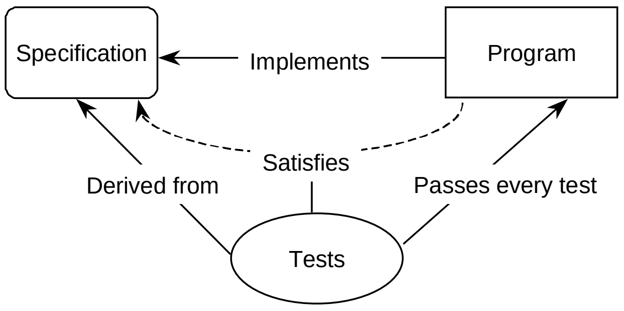


**Figure 2. Testing Model and Relationships**

In Figure 2, the specification symbol (i.e., rounded rectangle) is generically used to represent requirement, design, or test specifications. Certain specification languages have tool support that helps in developing complete and consistent specifications. Such tools provide the syntactic and semantic rigor that is required for transforming specifications into a form suitable for test vector generation. Model-based specification methods that support functional, state transition, and event-based techniques are increasing in popularity and use because the tool support has helped make them easier to use.[1]

A **model-based specification** approach constructs an abstract model of the system states and characterizes how a state is changed by abstract or concrete operations (paraphrased from Cohen et al. and Cook et al. [CHJ88; CGDDTK96]). Operations in the system are specified by defining the state changes or events that affect the model using existing mathematical constructs like sets or functions. **State transitions** define relationships between sequences of states based on conditions of the system state. **Event specifications** define certain conditions related to a change in the system state.

[1] Zave and Jackson [ZJ97] identify potential implementation bias of model-oriented techniques but support the claim that model-oriented techniques are gaining in popularity.

A **test specification model** is defined by a set of test specification elements, as shown in Figure 3. A **test specification element** is an input-to-output relation and an associated constraint defined by a conjunction (i.e., logically ANDed) of Boolean-valued relations that define constraints on the inputs associated with the input-to-output relation.

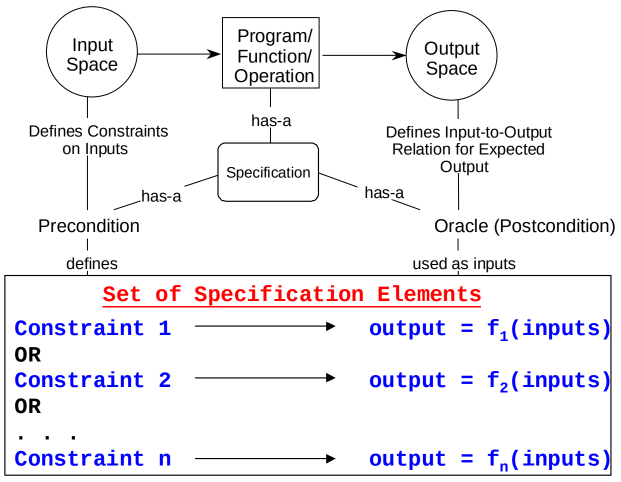

Input Space
Program/ Function/ Operation
Output Space
Defines Constraints on Inputs
has-a
Defines Input-to-Output Relation for Expected Output
Specification
has-a
has-a
Precondition
Oracle (Postcondition)
defines
used as inputs
Set of Specification Elements
Constraint 1 → output = f₁(inputs)
OR
Constraint 2 → output = f₂(inputs)
OR
. . .
Constraint n → output = fₙ(inputs)

**Figure 3. Representation of Test Specification Model**

Given a specification element, a **test vector** is a set of test inputs values derived from the constraint, and an expected output value derived from the input-to-output relation with respect to the test input values [BB96]. Informally, from a test generation perspective, a specification is **satisfiable** if at least one test vector exists for every specification element [BBF97].

## Model Transformation

Model transformations are typically required to transform model-based specifications into a form to support test generation. Hierons describes rewriting rules for Z specifications [Hie97] to support test case generation, but does not address specifications composed of combinations of specification techniques, particularly specifications composed using event specification techniques. In general, model transformation to support tool interoperability is an important area of investigation [Gil97].

Blackburn [Bla98] describes a tool-based approach for transforming a model-based specification into a form that supports test vector generation. The model-based specification supports composition using function, state, and event specifications. A translator implements rules for transforming SCR model specifications into a language used by the T-VEC test vector generation tool. The development of the prototype translator and evaluation environment helped identify shortcomings in the rules described in prior work that was presented at the 1997 Computer Assurance Conference [BBF97].

Similar model transformation efforts, not described in this paper, were performed for the ROOM method using the ObjecTime Developer toolset as part of the validation environment [BF98].

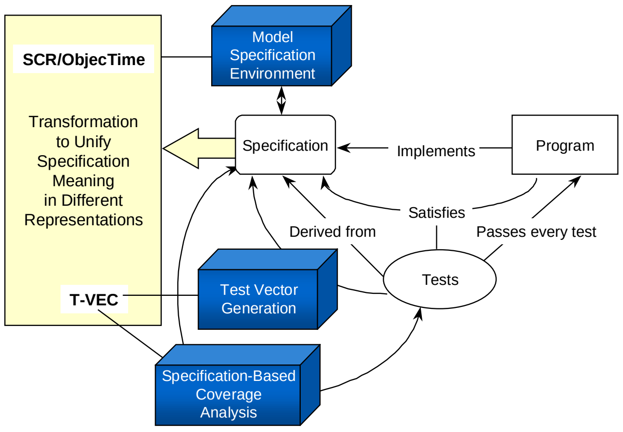
### *Evaluation Environment*

Figure 4 identifies generic tool types that are related to the elements of the test model shown in Figure 2. Such tools use or produce the three primary types of system artifacts (i.e., specifications, programs, and tests). A specific instance of this model was created to support the model transformation approach using the SCR tool (referred to as SCR* - pronounced SCR star) as the source for model-based specifications and the T-VEC tool system as the tool that supports test generation and specification-based coverage analysis.

***SCR****, developed by the Naval Research Laboratory, supports modeling and analysis of requirement specifications using a formal modeling language (i.e., a language with well-defined syntax and semantics).

***T-VEC,*** developed by T-VEC Technologies, Inc. supports:

- ***Test Vector Generation.*** A test vector generator produces test vectors from test specifications.
- ***Specification-Based Coverage Analysis.*** This tool analyzes the transformed specification to determine whether all specification elements have a corresponding test vector. This is the mechanism used to assess satisfiability of the transformed specifications.



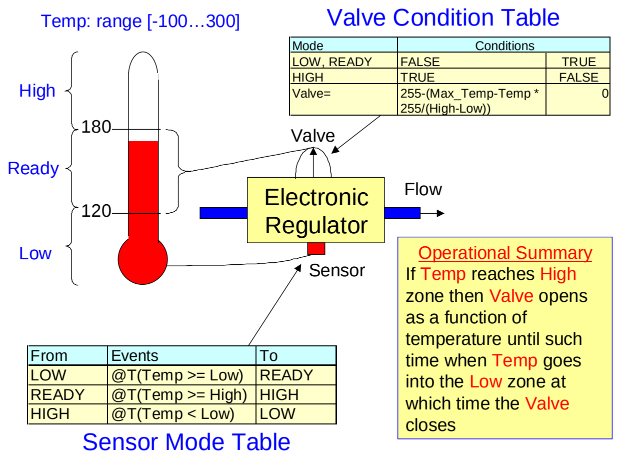
SCR/ObjecTime
Transformation to Unify Specification Meaning in Different Representations
T-VEC
Model Specification Environment
Specification
Implements
Program
Satisfies
Derived from
Passes every test
Tests
Test Vector Generation
Specification-Based Coverage Analysis

**Figure 4. Tools of the Evaluation Environment**

## Applications and Results

The remainder of this paper describes a simple example to illustrate the use of this approach for model analysis and testing. Consider the example shown in Figure 5 of an electronic regulator. The requirements for the regulator are:

- When the temperature reaches the High zone (i.e., 180 degress), the valve opens.
- The amount the valve opens is a function of the temperature from 120 degrees (closed) up to 300 degrees (fully open).
- Once the valve is open, it remains open until the temperature reaches the Low zone (i.e., 120 degrees).



Valve Condition Table

| Mode | Conditions | |
|---|---|---|
| LOW, READY | FALSE | TRUE |
| HIGH | TRUE | FALSE |
| Valve= | 255-(Max_Temp-Temp * 255/(High-Low)) | 0 |

| From | Events | To |
|---|---|---|
| LOW | @T(Temp >= Low) | READY |
| READY | @T(Temp >= High) | HIGH |
| HIGH | @T(Temp < Low) | LOW |

Sensor Mode Table

Operational Summary
If Temp reaches High zone then Valve opens as a function of temperature until such time when Temp goes into the Low zone at which time the Valve closes

**Figure 5. Example of Electronic Regulator**

The specification is described in the SCR tabular notation. Heitmeyer et al. [HJL96] describes the SCR method. The specification is defined in two parts. The **first part of the specification** defines the relationships between the temperature and the associated modes that relate to the temperature zones. This is referred to as the Sensor Mode Table shown in Figure 5. The system can be in one of three modes: LOW, READY, and HIGH. At the time when the temperature becomes greater than the constant Low (i.e., 120 degrees), the system transitions into the mode READY. The formal expansion of the event is:

```
@T(Temp >= Low) means:

if the previous value of Temp
denoted NOT(_Temp >= Low) and the
new value of Temp >= Low then the
event is true and the mode
transitions from LOW to READY
```

Table 1 shows the translated meaning for each event specification of the Sensor Mode Table. For each constraint, there is a minimal set of tests as shown in Table 2. The T-VEC test generation system uses a test selection heuristic based on domain testing theory where low-bound and high-bound values are selected for each constraint.[2] For example, the first test selects the low-bound value for the previous state value of _TEMP[3] (-100), which is less than the constant Low, and selects a value of 120 for the next state value of TEMP. For the high-bound selection, the value of 119 (i.e., one less that the constant Low) is selected for _TEMP, and 179 for TEMP (i.e., one less than the constant High).

**Table 1. Relationship of Translated Constraints**

| Events | Translation |
|---|---|
| @T(Temp >= Low) | (Temp >= Low) AND (  Temp < Low) |
| @T(Temp >= High) | (Temp >= High) AND (  Temp < High) |
| @T(Temp < Low) | (Temp < Low) AND (  Temp >= Low) |

**Table 2.  Tests for Each Translated Constraint**

| | Output | Inputs | | |
|---|---|---|---|---|
| Translation | Sensor | _Sensor | Temp | _Temp |
| (Temp >= Low) AND | READY | LOW | 120 | -100 |
| (_Temp < Low) | READY | LOW | 179 | 119 |
| (Temp >= High) AND | HIGH | READY | 180 | 120 |
| (_Temp < High) | HIGH | READY | 300 | 179 |
| (Temp < Low) AND | LOW | HIGH | -100 | 180 |
| (_Temp >= Low) | LOW | HIGH | 119 | 300 |

The **second part of the specification** defines the constraints and functions for the Value Condition Table shown in Figure 5. This table depends on the Sensor Mode Table. The Valve Condition Table is interpreted as follows:

```
if Sensor mode = High then
  Valve = 255-(Max_Temp-Temp
          * 255/(High-Low))
else if Sensor mode = LOW
     or Sensor mode = READY then
  Valve = 0
endif
```

Each SCR output variable and associated function map to a T-VEC functional relationship of an output variable with respect to the constraints on the input variables. The SCR model does not necessarily define a system state strictly in terms of constraints on the input variables as is required for T-VEC. For example, the Sensor mode is defined in terms of a mode transition table. This results in table dependencies as illustrated in Figure 6. The mode variables and the associated table relations must be transformed into constraints on the input variables.

[2] White and Cohen proposed domain testing theory as a strategy to select test points to reveal domain errors [WC80]. Their theory 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. When there is a strong correlation between the specification constraints and implementation paths, the selected test data should uncover computation and domain errors. As defined by Howden and refined later by Zeil, a computation error occurs when the correct path through the program is taken, but the output is incorrect due to faults in the computation along the path. A domain error occurs when an incorrect output is generated due to executing the wrong path through a program [How76; Zei89].

[3] An underbar (_) precedes the variable name to indicate that the variable represents the previous state variable before the event versus the next state variable after the event.

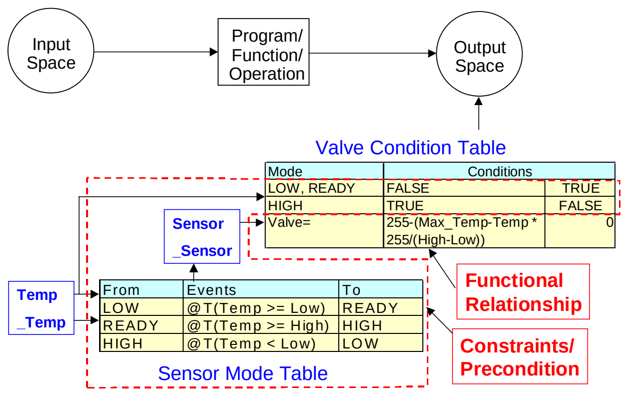


**Figure 6. Dependency Relationship**

Figure 6 provides a perspective of the example SCR specification represented in a form that is compatible with the test model shown in Figure 3. The constraint for the Valve Condition Table includes the conditions of the Valve table and the Sensor Mode transition table. This means that the constraint:

```
(Temp >= High) AND (_Temp < High)
```

must be satisfied (i.e., the Sensor mode is HIGH) as a requirement for the value to be computed using the functional relationship:

```
1)     255-(Max_Temp-Temp
2)     * 255/(High-Low))
```

In general, mode transition tables can have dependencies on other terms and modes. Events for modes and terms create the need to identify the previous and next state variable dependencies. As shown in Figure 6, the Sensor mode table depends on both the previous and next state input value of Temp; similarly the condition table Valve depends on the previous and next state variables Temp and Sensor.

A test specification requires the constraints of a specification to be defined strictly in terms of the input and output variables. A model-based approach defines states that are relations of inputs, terms, or state variables (e.g., Sensor, _Sensor). This allows the constraint/precondition and functional relationship (defined in terms of a Condition Table) to be defined as a relation on inputs, states, or terms. This approach typically simplifies the task of specifying behavior, but it is the key reason why a model transformation process is required.

### Static Analysis

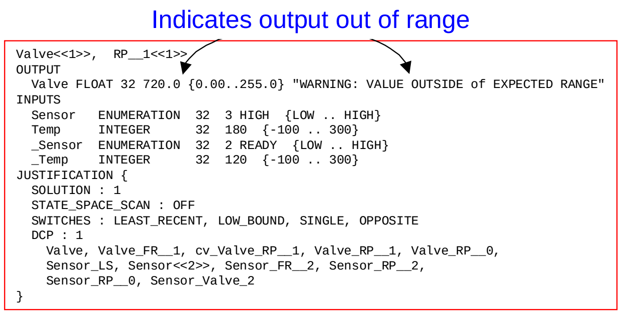
Static analysis helps determine whether there are contradictions in the model without executing the model. Contradictions exist if constraints cannot be satisfied. This is typically the most common problem, especially when the dependencies of specifications become large. This example is a simple 2-level dependency problem, but typical systems can have 10 or more dependency levels. It is also possible to identify functional relationships that specify values that are inconsistent with the domain of the output variables. These are analogous to computation errors in the code.

Consider the function to compute the Valve function. The requirements are that as the temperature reaches the maximum temperature (i.e., 300 degrees), the valve should be completely opened, and when the value reaches the constant Low (i.e., 120 degrees), the valve should be closed. Electronically controlled devices typically use some type of digital value to represent a fully open valve (in this case 255 – an 8-bit unsigned integer), and the value should be 0 when the valve is closed. It is common for implementors to make errors in scaled arithmetic conversions. To illustrate this point, the computation has two errors.

Figure 7 shows a sample test vector that has identified a problem in the computation. A warning is appended to the expected output because the computation is out of range. This is typically an indication that there is a computation error in the specification or that there are missing constraints on the inputs. The original expression (line 1 of the functional relationship under Figure 6) is missing parentheses around Max_Temp-Temp. In line 2, the subtraction should be Max_Temp-Low rather than High-Low. The correct computation is as follow:

```
255-((Max_Temp-Temp) *
255/(Max_Temp-Low))
```

Identifying this type of problem is time consuming. In addition, it is well known that identification and removal of errors in the implementation or integration phase is much more costly than it is during the requirements phase.


Indicates output out of range

```
Valve<<1>>,  RP__1<<1>>
OUTPUT
  Valve FLOAT 32 720.0 {0.00..255.0} "WARNING: VALUE OUTSIDE of EXPECTED RANGE"
INPUTS
  Sensor   ENUMERATION  32  3 HIGH  {LOW .. HIGH}
  Temp     INTEGER      32  180  {-100 .. 300}
  _Sensor  ENUMERATION  32  2 READY  {LOW .. HIGH}
  _Temp    INTEGER      32  120  {-100 .. 300}
JUSTIFICATION {
  SOLUTION : 1
  STATE_SPACE_SCAN : OFF
  SWITCHES : LEAST_RECENT, LOW_BOUND, SINGLE, OPPOSITE
  DCP : 1
    Valve, Valve_FR__1, cv_Valve_RP__1, Valve_RP__1, Valve_RP__0,
    Sensor_LS, Sensor<<2>>, Sensor_FR__2, Sensor_RP__2,
    Sensor_RP__0, Sensor_Valve_2
}
```

**Figure 7. Internal Form of Test Vector With Warning**

Figure 8 provides a summary of a minimal set of test values for the translated condition table for Valve. In this figure, the associated test selection mode (i.e., LOW_BOUND, HIGH_BOUND) is also shown.

| | | | Output | Input | | | | Test Selection |
|---|---|---|---|---|---|---|---|---|
| Mode | Conditions | | Valve | Sensor | _Sensor | Temp | _Temp | Mode |
| HIGH | TRUE | FALSE | 85.00 | HIGH | READY | 180 | 120 | LOW_BOUND |
| | | | 255.00 | HIGH | READY | 300 | 179 | HIGH_BOUND |
| LOW, READY | FALSE | TRUE | 0.00 | LOW | HIGH | -100 | 180 | LOW_BOUND |
| | | | 0.00 | LOW | HIGH | 119 | 300 | HIGH_BOUND |
| | | | 0.00 | READY | LOW | 120 | -100 | LOW_BOUND |
| | | | 0.00 | READY | LOW | 179 | 119 | HIGH_BOUND |

**Figure 8. Test Vectors for Valve Condition Table**

### *Sample Results*

Table 3 shows some sample results on the application of this approach to other systems. Each specification originally had one or more specification problems or anomalies. As seen in Figure 5, the electronic regulator problem is very small (two tables, five functional relationship, two constraints, and maximum depth of two table dependencies). A flight guidance system is a real-world industrial problem [Mil98]; it has 78 tables, 423 functional relationships, 7,349 constraints, and a maximum dependency depth of 12. The results on this project are planned for publication in the next year.

**Table 3. Sample Results Statistic**

| System/Projects | Condition Table | Event Table | Mode Table | Functional Relationship | Constraint | Level |
|---|---|---|---|---|---|---|
| Temperature regulator | 1 | | 1 | 5 | 6 | 2 |
| Safety injection | 1 | 1 | 1 | 10 | 68 | 3 |
| Electronic flight instrumentation system | 37 | 5 | 0 | 88 | 389 | 3 |
| Elevator system | 10 | 6 | 0 | 38 | 90 | 3 |
| Flight guidance system | 49 | 15 | 14 | 423 | 7349 | 12 |

## Summary

Software testing will play a role in the development of software systems for some time to come. Although testing can account for 40 to 75 percent of the lifetime development and maintenance costs, the results summarized in this paper provide promising evidence that the use of test automation to support the manually intensive test generation and model-based analysis is feasible and practical.

There is a great need to demonstrate and integrate new and advanced technologies. This paper describes an environment developed to validate the use of model-based translators on real-world applications. The environment integrates model-based development tools with a specification-based test vector generator and specification-based coverage analyzer.

As modeling tools and associated methodologies continue to evolve, these results provide the basis for building translators for other modeling tools. This allows new tooling technology to be integrated with existing tools and has the indirect effects of reducing the cost and time of specialized training and tool expenditures.

The ability to integrate front-end development tools with back-end testing tools fosters the use of model-development tools, and such tools can significantly reduce the maintenance phase of a product, which typically consumes 70 percent of the product life cycle. Maintenance typically requires minimal development effort but typically large efforts in testing. Because the original developers usually are not available to assist in maintenance and evolution efforts, test automation can significantly minimize reverification efforts because the designer's requirement and design knowledge is captured in model specifications.

## References

[BB96] Blackburn, M.R., R.D. Busser, T-VEC: A Tool for Developing Critical System. In Proceeding of the Eleventh International Conference on Computer Assurance, Gaithersburg, Maryland, pages 237-249, June, 1996.

[BBF97] Blackburn, M.R., R.D. Busser, J.S. Fontaine, Automatic Generation of Test Vectors for SCR-Style Specifications, In Proceeding of the 12th Annual Conference on Computer Assurance, Gaithersburg, Maryland, pages 54-67, June, 1997.

[Bei83] Beizer, B. Software Testing Techniques, New York, New York: Van Nostrand Reinhold, 1983.

[BF98] Blackburn, M.R., J.S. Fontaine, Specification Transformation to Support Automated Testing, TR SPC-97036-MC, Version 02.00.01, Software Productivity Consortium, March 1998.

[Bla98] Blackburn, M.R., Specification Transformation and Semantic Expansion to Support Automated Testing, Ph.D. Dissertation, George Mason University, 1998.

[CGDDTK96] Cooke, D., A. Gates, E. Demirors, O. Demirors, M. Tankik, B. Kramer, Languages for the Specification of Software, Journal of Systems Software, 32:269-308, 1996.

[CHJ88] Cohen, B., W. T. Harwood, M.I. Jackson, The Specification of Complex System, Addison-Wesley, Great Britain, 1988.

[GG75] Goodenough, J. B., S. L. Gerhart, Toward a Theory of Test Data Selection, IEEE Transactions on Software Engineering, 1(2):156-173, 1975.

[Gil97] Gill, D. H., Formal Methods for Software Evolution, Solicitation, Defense Advanced Research Projects Agency, BAA 98-10, November, 1997.

[Gou83] Gourlay, J.S., Introduction to the Formal Treatment of Testing, Software Validation. Proceeding of the Symposium on Software Validation, 1983.

[GW94] Ghiassi, M., K.I.S. Woldman, Dual Programming Approach to Software Testing, Software Quality Journal, 3:45-58, 1994.

[Hen80] Heninger, K., Specifying Software Requirements for Complex Systems: New Techniques and Their Application, IEEE Transactions on Software Engineering, 6(1):2-13, 1980.

[Hie97] Hierons, R. M. Testing from a Z Specification, Journal of Software Testing, Verification and Reliability, 7:19-33, 1997.

[HJL96] Heitmeyer, C., R. Jeffords, B. Labaw, Automated Consistency Checking of Requirements Specifications. ACM TOSEM, 5(3):231-261, 1996.

[How76] Howden, W.E., Reliability of the Path Analysis Testing Strategy, IEEE Transactions on Software Engineering, 2(9):208-215, 1976.

[Mil98] Miller, S., Specifying the Mode Logic of a Flight Guidance System in CoRE and SCR. Accepted to the Second Workshop on Formal Methods in Software Practice (FMSP'98), Clearwater Beach, Florida, March, 1998.

[SGW94] Selic, B., G. Gullekson, P.T. Ward, Real-Time Object-Oriented Modeling. New York, New York: John Wiley & Sons, Inc, 1994.

[SPC93] Software Productivity Consortium, Consortium Requirements Engineering Guidebook, SPC-92060-CMC, version 01.00.09. Herndon, Virginia, 1993.

[WC80] White, L.J., E.I. Cohen, A Domain Strategy for Computer Program Testing. IEEE Transactions on Software Engineering, 6(3):247-257,May, 1980.

[Zei89] Zeil, S.J., Perturbation Techniques for Detecting Domain Errors, IEEE Transactions on Software Engineering, 15(6):737-746, 1989.

[ZJ97] Zave, P., M. Jackson, Four Dark Corners of Requirements Engineering, ACM Transactions on Software Engineering and Methodology, 6(1):1-30, 1997.