Mars Polar Lander Fault Identification Using Model-based Testing

Mark Blackburn
Robert Busser
Aaron Nauman
Software Productivity Consortium/T-VEC
2214 Rock Hill Road, Herndon, VA 20170

Robert Knickerbocker
Richard Kasuda
Lockheed Martin Space Systems Company
Astronautics Operations
P.O. Box 179, Denver, CO 80201-0179

Abstract

This paper describes the application of the Test Automation Framework (TAF) on the Mars Polar Lander (MPL) software. The premature shutdown of the descent engine on the MPL spacecraft is believed to be the most likely cause for the mission failure. It is believed that the engine shutdown occurred when the three landing legs were extended into their deployed position. This event created an unanticipated transient touchdown indication from the legs, causing the software to inadvertently shutdown the descent engines prior to reaching the surface of Mars. This spurious indication should have been ignored by the Touchdown Monitor (TDM) software, but due to a design flaw, was actually "latched," thus causing the premature engine shutdown. The TAF approach was used to model the TDM software requirements. The associated TAF tools generated tests that identified a potential TDM fault.

1. Introduction

The Mars Polar Lander (MPL) was launched on January 3, 1999 and lost on December 3, 1999. A premature shutdown of the MPL descent engine is believed to be the most likely cause for the mission failure. It is believed that the engine shutdown occurred because of a failure to properly process an electrical transient when the three landing legs were extended into their deployed position. This event created an incorrect touchdown indication from the legs, causing the software to inadvertently shutdown the descent engines prior to reaching the surface of Mars. This spurious indication should have been ignored by the Touchdown Monitor software, but due to a design flaw, was actually "latched," thus causing the premature engine shutdown. Lockheed Martin Space Systems Company Astronautics Operations (LMAO - Denver) was responsible for the development and verification of the MPL spacecraft and on board software.

The Software Productivity Consortium (Consortium) members have requested greater support in the area of verification and test automation. To address the need the Consortium developed capabilities referred to as the Test Automation Framework (TAF). TAF integrates various government and commercially available model development and test generation tools to support defect prevention and automated testing of systems and software. The Consortium helps members with technology transfer through pilot projects. As a result LMAO requested that the TAF team use TAF to detect the error in the TDM system.

The objective was to demonstrate the capability of the TAF approach to detect a deeply hidden problem in the implementation of an MPL component called the Lander Touchdown Monitor (TDM). LMAO sent the requirements and the code to the Consortium TAF team, but did not disclose the source or location of the problem. At that time the TAF team was not aware of the details of the TDM problem. This paper provides an overview of the results of applying the TAF to the TDM problem.

1.1 Results summary

The TAF team developed a model for the TDM system from the requirements supplied by LMAO using the Software Cost Reduction (SCR) tool [HJL96]. The TAF team spent about 12 staff hours modeling the requirements and building the test driver schema to support test injection into the TDM C code module supplied by the LMAO team. The TDM module is approximately 50 source lines of code. The team generated test vectors and test drivers using the T-VEC system. The tests were injected into the TDM code module in an attempt to uncover the fault. The problem was not identified.

The TAF team flew to LMAO (Denver) to present their results on the following week. The developer of the code, along with the TDM team and other related LMAO V&V staff, were present at the presentation. The TAF team explained the model, and executed the generated tests against the code. The TAF team observed that the TDM code used state data that was managed and accessed through several code entry points, and the TDM developer confirmed this. The TAF team modified the test schema to simulate multiple calls to the entry points much like that of the real-time multi-tasking executive of the TDM software. These additional calls would propagate the state data. New test drivers were automatically generated from the original model, and the modified test drivers were executed exposing the fault.

The T-VEC test generation system uses a test selection heuristic based on **domain testing theory** [WC80] where test values are selected for each constraint. Domain testing theory is based on the intuitive idea that faults in implementation are

more likely to be found by test points chosen near appropriately defined program input and output domain boundaries [TVK90]. The test vectors stimulating the failures were associated with the constraint that represents the situation where a landing leg sensor indicates touchdown for two consecutive reads. The test driver generation mechanism provides the flexibility to simulate the periodic calls of the TDM real-time executive. The combination of the test vector generation, which selects inputs for the critical constraint, and test driver generation, which emulates the real-time executive provided the stimulus to initiate a test failure associated with the probable program fault.

The results of the application suggest that the TAF approach may have the potential to provide a systematic and cost-effective approach for verification. LMAO believes the tool provides a standardized test approach and a more thorough test capability than the manual approach. LMAO and its customers are considering future pilot projects to more fully assess the TAF capabilities.

2. Approach and toolset

2.1 Process overview

The conceptual process flow that relates the artifacts to the tools is shown in Figure 1. The TDM specification is modeled using the SCRtool. An SCR-to-T-VEC translator translates the SCR model to a T-VEC test specification. T-VEC automatically generates test vectors (i.e., test cases with test input values, expected output values and traceability information) and requirement-to-test coverage metrics. T-VEC automatically generates test drivers to execute tests against the TDM code compiled in a Microsoft C++ development environment running on a Windows NT platform. The execution of the test driver results in actual outputs that are then compared with the expected outputs, and the results report is produced. SCR concepts and tool

2.2 SCR concepts and tool

SCR is a table-based modeling approach that models system and software requirements. SCR represents system inputs as **monitored variables**, system outputs as **controlled variables** and intermediate values as **term variables**. Variables are defined as primitive types (e.g., Integers, Float, Boolean, Enumeration) or as user-defined types. Behavior is defined using a tabular approach relating four model elements: modes, conditions, events, and terms. The required functionality or behavior of the system is defined using tables to relate monitored variables to controlled variables. There are three basic types of tables (with two variants):

- Condition table (with mode or modeless)
- Event table (with mode or modeless)
- Mode transition table for a mode class

A **mode class** is a state machine, where system states are called system modes and the transitions of the state machine are characterized by guarded events. A **condition** characterizes

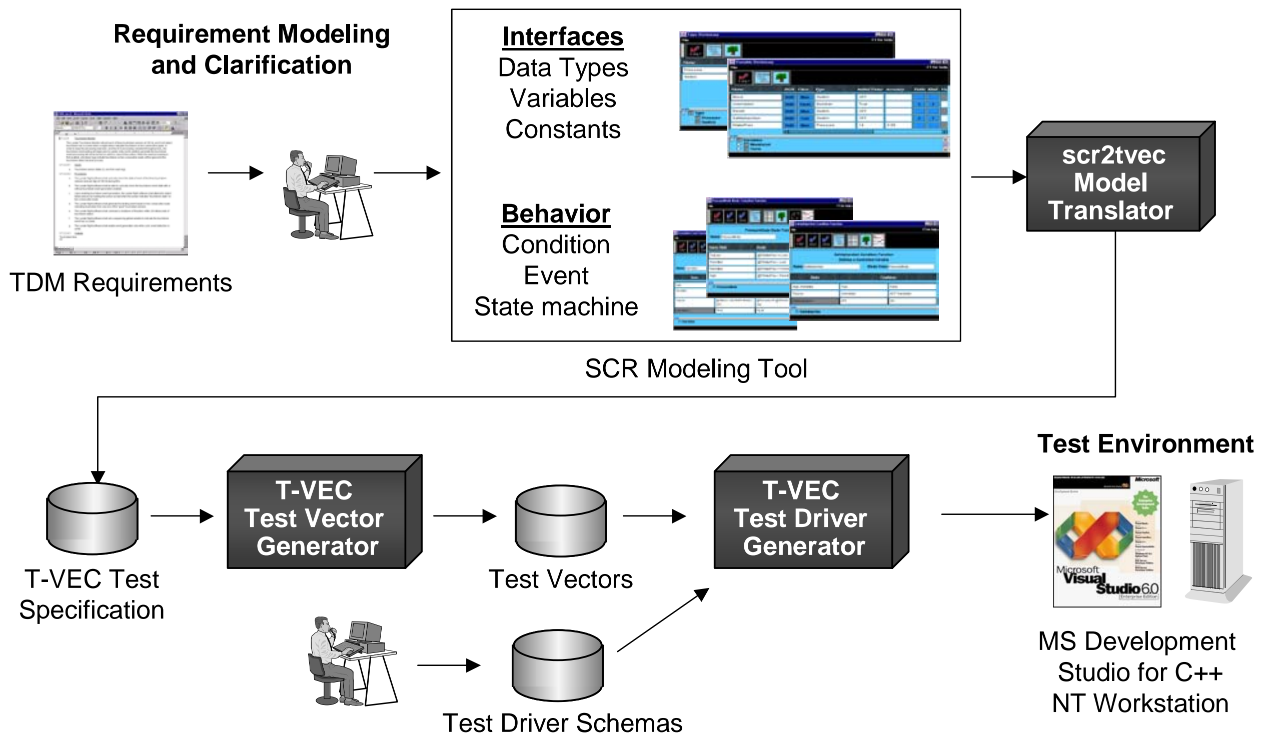


Figure 1. Process flow and artifacts

system state with an expression that evaluates to true or false. An **event** occurs when any system entity changes value.

The SCR modeling approach permits condition, event and mode tables to be combined. Terms and controlled variables are functions of input variables, modes, or other terms. Their values are defined in the model through event or condition tables. This allows complex relationships between monitored and controlled variables to be described using terms with simpler relationships modeled in condition, event or mode tables.

3. TDM requirements and model

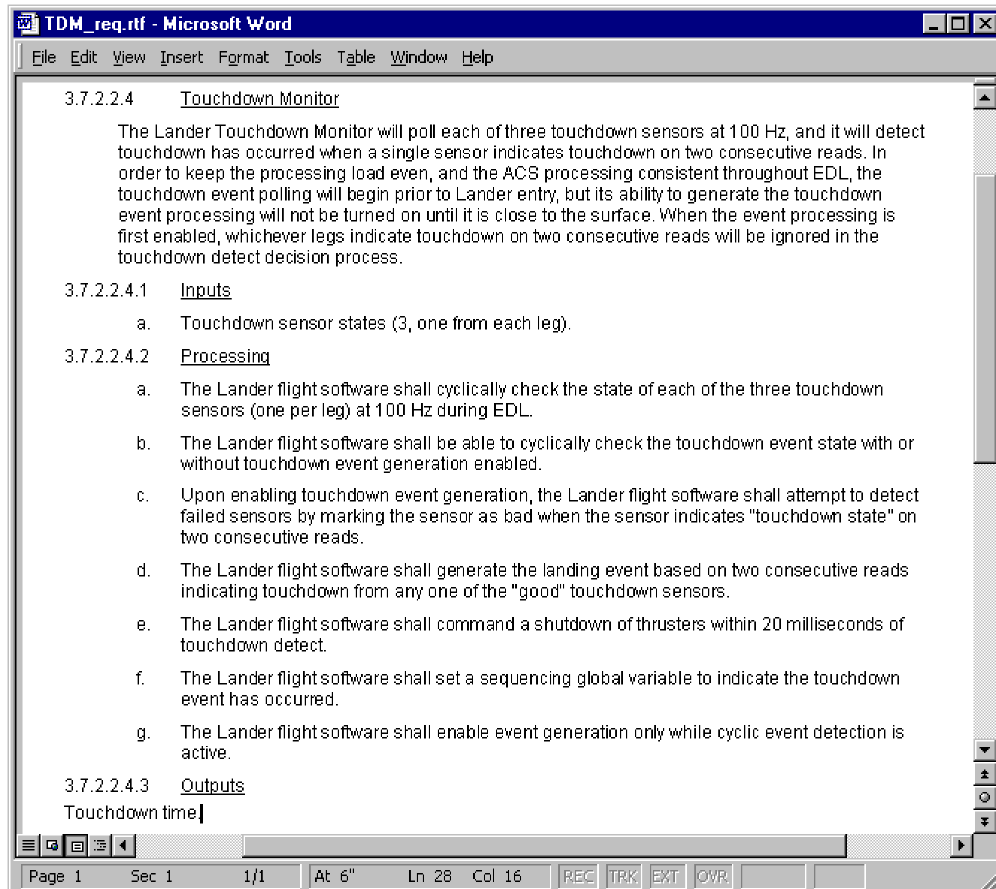
The LMAO TDM team supplied the textual requirements shown in Figure 2 to the TAF Team.

3.1 Requirements analysis

Developing SCR models requires identifying the system monitored (input) and controlled (output) variables, and defining the relationships between them. This process is typically iterative. It involves defining the variables, data types associated with the variables, and the tables that define relationships between the variables. A useful guideline for developing SCR models is to work backwards from each output to make the process goal-oriented. The value of each output is defined in terms of the system inputs. Term variables are introduced whenever intermediate values are necessary or useful. Breaking the TDM requirement into clauses supports identifying variables and relationships. Table 1 contains elaboration and clarification of the TDM requirements to support modeling. It identifies the variables and relations associated with each clause.

The monitored (input) variables identified in the system can be refined into the following set:

- TD_1, TD_2, TD_3 – the current sensor value for landing legs 1, 2, and 3 respectively
- TD_1_Last, TD_2_Last, TD_3_Last - the sensor value for landing legs 1, 2, and 3 from the previous cycle
- CMD_disable_enable – the state of the event generation flag – when enabled the touchdown signal can be issued
- TDM_started – the global variable that allows the TDM executive to run



TDM_req.rtf - Microsoft Word

3.7.2.2.4 Touchdown Monitor

The Lander Touchdown Monitor will poll each of three touchdown sensors at 100 Hz, and it will detect touchdown has occurred when a single sensor indicates touchdown on two consecutive reads. In order to keep the processing load even, and the ACS processing consistent throughout EDL, the touchdown event polling will begin prior to Lander entry, but its ability to generate the touchdown event processing will not be turned on until it is close to the surface. When the event processing is first enabled, whichever legs indicate touchdown on two consecutive reads will be ignored in the touchdown detect decision process.

3.7.2.2.4.1 Inputs

a. Touchdown sensor states (3, one from each leg).

3.7.2.2.4.2 Processing

a. The Lander flight software shall cyclically check the state of each of the three touchdown sensors (one per leg) at 100 Hz during EDL.

b. The Lander flight software shall be able to cyclically check the touchdown event state with or without touchdown event generation enabled.

c. Upon enabling touchdown event generation, the Lander flight software shall attempt to detect failed sensors by marking the sensor as bad when the sensor indicates "touchdown state" on two consecutive reads.

d. The Lander flight software shall generate the landing event based on two consecutive reads indicating touchdown from any one of the "good" touchdown sensors.

e. The Lander flight software shall command a shutdown of thrusters within 20 milliseconds of touchdown detect.

f. The Lander flight software shall set a sequencing global variable to indicate the touchdown event has occurred.

g. The Lander flight software shall enable event generation only while cyclic event detection is active.

3.7.2.2.4.3 Outputs

Touchdown time

Figure 2. TDM requirements

Table 1. TDM requirements

Requirement Statement/Clause	Variables	Relations
TDM(a) The Lander flight software shall cyclically check the state of each of the three touchdown sensors (one per leg) at 100 Hz during EDL.		Periodic processing controlled in test driver
TDM(b) The Lander flight software shall be able to cyclically check the touchdown event state with or without touchdown event generation enabled.	TD_1, TD_1_Last TD_2, TD_2_Last TD_3, TD_3_Last	TD_Sen1, TD_Sen2, TD_Sen3
TDM(c) Upon enabling touchdown event generation, the Lander flight software shall attempt to detect failed sensors by marking the sensor as bad when the sensor indicates "touchdown state" on two consecutive reads.	All	First_Marked_Bad
TDM(d) The Lander flight software shall generate the landing event based on two consecutive reads indicating touchdown from any one of the "good" touchdown sensors.	All	TDM_thruster
TDM(e) The Lander flight software shall command a shutdown of thrusters within 20 milliseconds of touchdown detect.		Outside scope of code module
TDM(f) The Lander flight software shall set a sequencing global variable to indicate the touchdown event has occurred.	All	TDM_thruster
TDM(g) The Lander flight software shall enable event generation only while cyclic event detection is active.	CMD_disable_enable TDM_started	TDM_event_enabled, TMD_Modes, TDM_thruster

Although the requirements document indicates that the output is "Touchdown time," the key output associated with the code is called "TDM_thruster" which is modeled as an enumerated data type that can take on the value of DISABLE – meaning that the thruster is shut off, or ENABLE, meaning that the thruster is on:

- TDM_thruster – the variable associated with the control of the TDM thruster

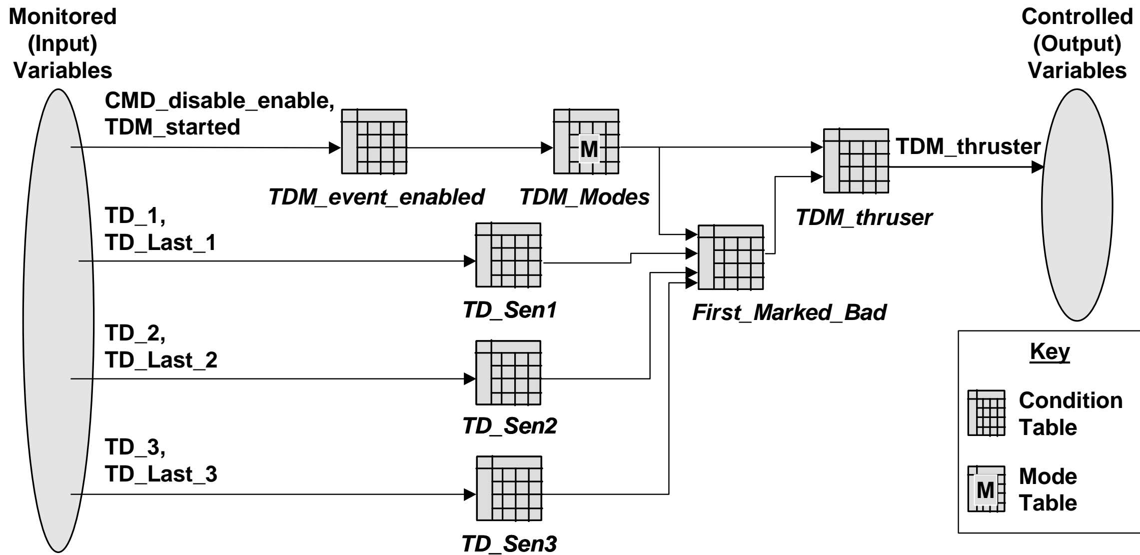


Figure 3. Model structure for TDM

3.2 Modeling functional requirements

Once the system's data is defined, its behavior can be modeled. In SCR, this involves defining the values of the controlled (output) variables through condition, event, or mode tables. These tables define the value of a variable in terms of monitored (input), terms (intermediate), and mode (state) variables. Figure 3 provides a representation of the TDM model. A condition table defines the output value for TDM_thruster. It depends on five term tables and one mode table. These term and mode tables are directly associated with the relations defined in Table 1. They result from relationships derived from the textual requirements. A value of a **term variable** is defined through a condition or event

table as an intermediate value. Terms can be referenced in the constraints or value calculations of other terms or controlled variables. They reduce the model complexity by simplifying expressions and eliminating redundancies.

The TDM(b) requirement results in the three terms TD_Sen1, TD_Sen2, and TD_Sen3 that define the conditions associated with the sensor signal for each landing leg. They are related to the requirement TDM(c) through the term First_Marked_Bad. This First_Marked_Bad term models the requirement for detecting a failed sensor, where the first sensor with two consecutive reads is marked bad. The term First_Marked_Bad also depends on TDM_Modes, which depends on TDM_event_enabled. These terms represent conditions and states associated with enabling event generation. The combination of these term variables are used to represent the requirements for TDM(d) and TDM(f) that define the values of the output TDM_thruster. The model details are described in the following sections, and Figure 4 provides the detailed tabular specification for the term and condition variables.

3.3 Modeling relations TD_Sen1, TD_Sen2, and TD_Sen3

The term TD_Sen1 defines the conditions when the touch down sensor indicates that two consecutive reads have occurred for landing leg one. When the conditions TD1 and TD_Last_1 (previous read) are satisfied the value of TD_Sen1 is TRUE, otherwise it is FALSE. The relations for TD_Sen2 and TD_Sen3 are similar to TD_Sen1, but apply to landing legs 2 and 3 respectively.

3.4 Modeling relation TDM_event_enable

The term TDM_event_enable is an Enumerated variable. The specified value for TDM_event_enable is TDM_YES when the command to enable event generation has been enabled (CMD_disable_enable = ENABLE) and the TDM executive had been allowed to run (TDM_started = TDM_YES), otherwise TDM_event_enabled is set to TDM_NO.

3.5 Modeling relation TDM_modes

The mode TDM_Modes defines the event when the TDM software transitions from the state Before_event to the Event_gen state. This occurs at the time when TDM_event_enabled takes on the value TDM_YES (and is represented by the event expression @T(TDM_event_enabled = TDM_YES). The mode table also defines the event transition for transitioning from the Event_gen state back to the Before_event state when the TDM_event_enabled changes to the value TDM_NO.

3.6 Modeling relation first_marked_bad

The term First_Marked_Bad is modeled as an Integer that returns a value between zero and three. The table First_Marked_Bad is also associated with the mode table TDM_Modes. The first column of the table contains the value Before_event and Event_gen, which are the two possible modes for TDM_Modes. These mode values are combined with the conditions as they specify the required value for the output First_Marked_Bad. When the mode is Before_event the value of First_Marked_Bad must always be zero, as indicated by the TRUE condition in the row associated with Before_event mode. When the mode is Event_gen, the value of First_Marked_Bad takes on the value of one, two or three depending on the condition associated with the term for the sensors TD_Sen1, TD_Sen2, or TD_Sen3, otherwise it takes on the value zero.

3.7 Modeling relation TDM_thruster

The condition table for TDM_thruster also is shown in Figure 4. Like First_Marked_Bad, TDM_thruster is also associated with the mode table TDM_Modes. When the mode is Before_event the thruster must always be ENABLE. After the Event_gen, the thruster takes on the value DISABLE when TDM_started is equal to TDM_YES, with one of three possible conditions:

1. First_Marked_Bad = 1, indicating that sensor leg 1 has been marked bad, and then sensor leg 2 (TD_Sen_2) or sensor leg 3 (TD_Sen_3) has become true.
2. First_Marked_Bad = 2, indicating that sensor leg 2 has been marked bad, and then sensor leg 1 (TD_Sen_1) or sensor leg 3 (TD_Sen_3) has become true.
3. First_Marked_Bad = 3, indicating that sensor leg 3 has been marked bad, and then sensor leg 1 (TD_Sen_1) or sensor leg 2 (TD_Sen_2) has become true.

Otherwise, if the mode is still Event_gen, then TDM_thruster must be ENABLE when:

1. First_Marked_Bad is zero – indicating that no sensor has been activated
2. First_Marked_Bad is 1, but neither sensor for leg 2 or 3 has been sensed
3. First_Marked_Bad is 2, but neither sensor for leg 1 or 3 has been sensed
4. First_Marked_Bad is 3, but neither sensor for leg 1 or 2 has been sensed

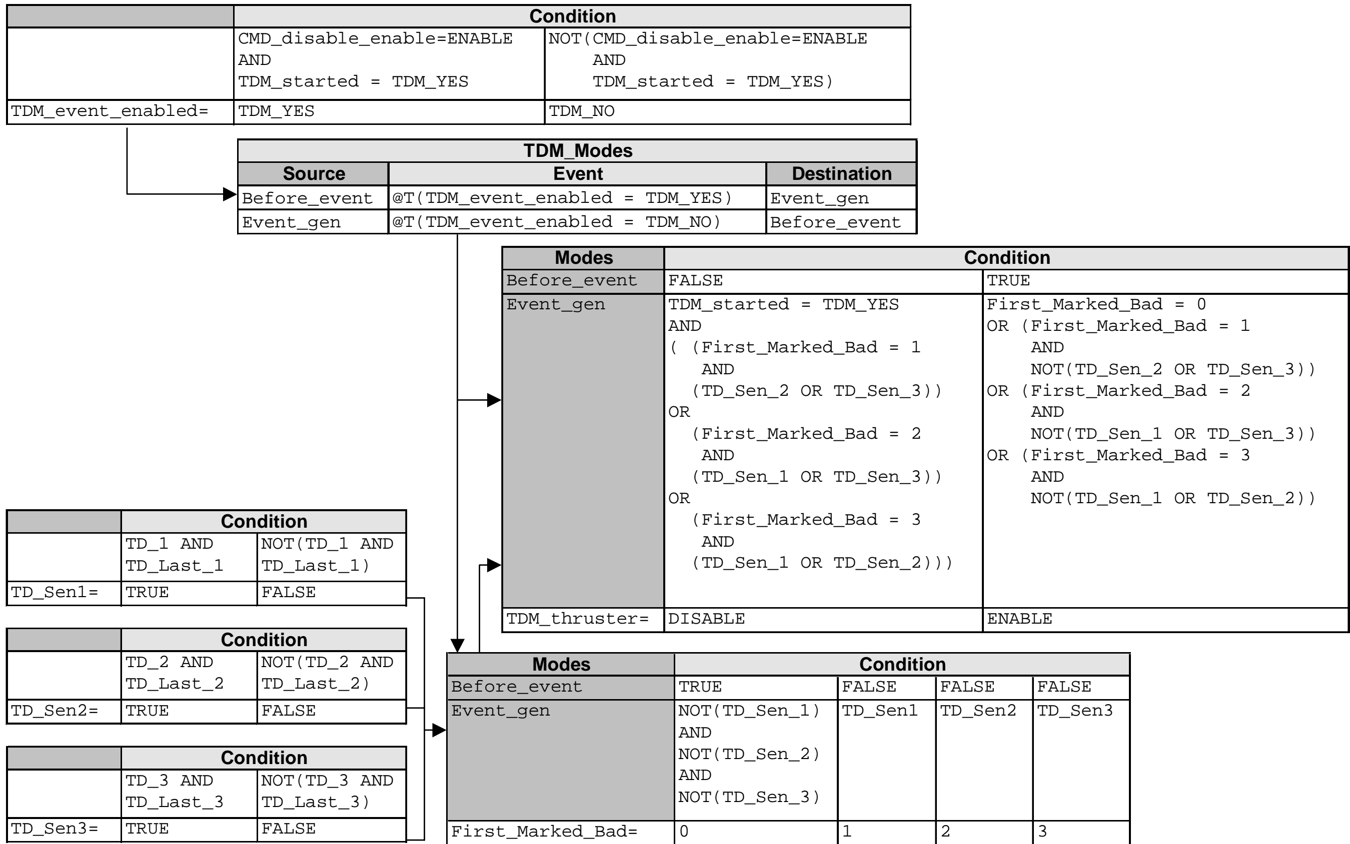


Figure 4. Behavioral specifications for TDM

Table 2. Test vectors for TDM_thruster

Test ID		Controlled (Output)	Monitored (Inputs)									
Vector #	DCP	TDM_thruster	TDM_modes	TDM_event_enabled	CMD_disable_enable	TD_1	TD_Last_1	TD_2	TD_Last_2	TD_3	TD_Last_3	First_Marked_Bad
1	1	ENABLE	Before_event	TDM_NO	DISABLE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	3
2	1	ENABLE	Before_event	TDM_NO	DISABLE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	0
3	2 3	DISABLE	Event_gen	TDM_YES	ENABLE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	1
4	2	DISABLE	Event_gen	TDM_YES	ENABLE	TRUE	TRUE	TRUE	TRUE	FALSE	FALSE	1
5	3	DISABLE	Event_gen	TDM_YES	ENABLE	TRUE	TRUE	FALSE	FALSE	TRUE	TRUE	1
6	4 5	DISABLE	Event_gen	TDM_YES	ENABLE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	2
7	4	DISABLE	Event_gen	TDM_YES	ENABLE	TRUE	TRUE	TRUE	TRUE	FALSE	FALSE	2
8	5	DISABLE	Event_gen	TDM_YES	ENABLE	FALSE	FALSE	TRUE	TRUE	TRUE	TRUE	2
9	6 7	DISABLE	Event_gen	TDM_YES	ENABLE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	3
10	6	DISABLE	Event_gen	TDM_YES	ENABLE	TRUE	TRUE	FALSE	FALSE	TRUE	TRUE	3
11	7	DISABLE	Event_gen	TDM_YES	ENABLE	FALSE	FALSE	TRUE	TRUE	TRUE	TRUE	3
12	8	ENABLE	Event_gen	TDM_YES	ENABLE	FALSE	TRUE	FALSE	TRUE	FALSE	TRUE	0
13	8	ENABLE	Event_gen	TDM_YES	ENABLE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	0
14	9	ENABLE	Event_gen	TDM_YES	ENABLE	TRUE	TRUE	FALSE	TRUE	FALSE	TRUE	1
15	9	ENABLE	Event_gen	TDM_YES	ENABLE	TRUE	TRUE	FALSE	FALSE	FALSE	FALSE	1
16	10	ENABLE	Event_gen	TDM_YES	ENABLE	FALSE	TRUE	TRUE	TRUE	FALSE	TRUE	2
17	10	ENABLE	Event_gen	TDM_YES	ENABLE	FALSE	FALSE	TRUE	TRUE	FALSE	FALSE	2
18	11	ENABLE	Event_gen	TDM_YES	ENABLE	FALSE	TRUE	FALSE	TRUE	TRUE	TRUE	3
19	11	ENABLE	Event_gen	TDM_YES	ENABLE	FALSE	FALSE	FALSE	FALSE	TRUE	TRUE	3

4. Test vector generation

The SCR-to-T-VEC model translator transforms each SCR table into a T-VEC subsystem. The T-VEC compiler converts each subsystem into a set of primitive test specifications for test vector generation [BBF97]. The translated and compiled version of the TDM_thruster requirement includes 11 test specifications. The test vector generator attempts to determine two test vectors for each test specification based on a test selection strategy derived from the concept of domain testing theory. The test generation produced 22 test vectors, one for the low-bound and one for the high-bound values of the 11 test specifications. Table 2 shows a tabular representation of the 19 unique test vectors produced for TDM_thruster. Test vectors with duplicate input and output values derived from different test specifications are reduced into a unique set of test vectors. The test vectors include 10 monitored variables and 7 term variables (not shown in the table). The test values shown in Table 2 reflect how the test generator systematically selects low-bound and high-bound test points at the domain boundaries. The input value ranges and constraints (e.g., relational operators) of the specification define the domain boundaries. For example, vector # 1, First_Marked_Bad = 3 is based on high-bound values of the data type range of SensorIDType, while vector # 2, First_Marked_Bad = 0 is based on the low-bound for the data type range. The remainder of the test cases, specifically test vectors three through 19 are associated with the TDM_modes = Event_gen. These cases systematically cover the cases for each combination of the sensor values for legs one, two, and three, for the various possible situations when a different sensor leg is marked bad.

5. Test driver generation and execution

The last step in the process produces test drivers that execute against the code. The test driver generator combines test driver schemas, user-defined object mappings and test vectors to produce test drivers as illustrated in Figure 5. The test driver schema encodes an algorithmic pattern for test execution for the specific test environment. The object mappings relate model variables to the implementation objects. The test driver generator creates test drivers by repeating the execution steps defined in the schema for each test vector. There are typically four primary steps for executing each test case:

- Set the value of the test output to some value other than what is expected
- Set the values of the test inputs
- Cause execution of the test
- Retrieve and save the results of the test execution

Test driver schemas describe how to accomplish these steps for a specific testing environment using a small language that accesses information about the specification model, data objects, types, ranges, test values, and user customizable information. A schema also describes the form of expected outputs to support results analysis.

An existing C test driver schema was used to produce the test driver file TDM_thruster.c, which is the main program for the test. TDM_thruster.c is compiled and linked with sam_Touchdown_Monitor.c (the actual C module for the TDM software). The test driver TDM_thruster.c performs some initialization, sets the inputs, calls the subsystem under test, and stores the resulting output.

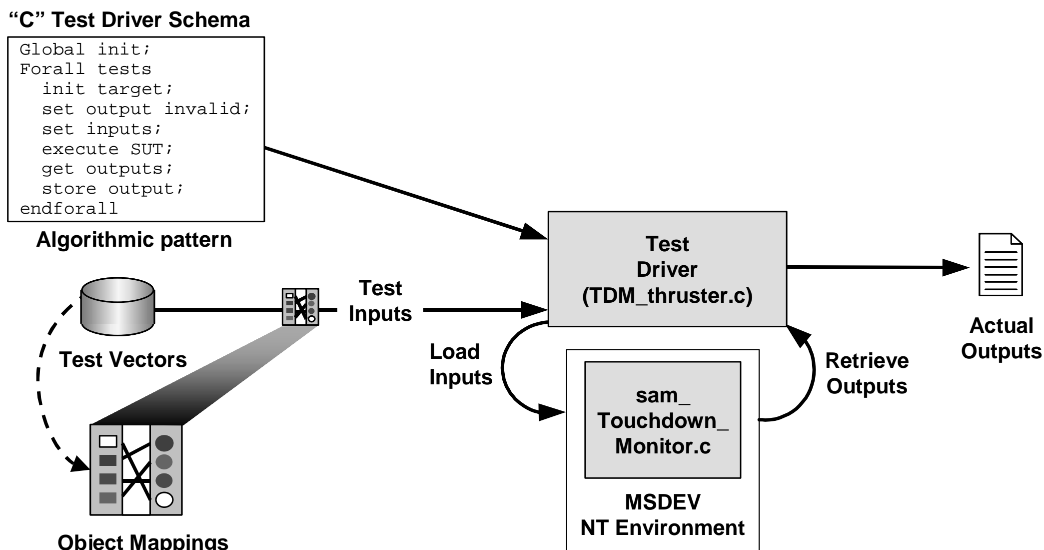


Figure 5. Elements of a test driver

The generated test drivers were executed on a Windows NT platform. The initial execution with the original test driver model did not identify any failure, because the test driver did not operate like the actual code. A real-time multi-tasking executive periodically calls the TDM code. Continuous calls to the entry point can propagate and latch state data. The latched data can inadvertently signal a shutdown of the thruster.

The original test driver called the main entry point once per test vector, and could not propagate the state data that caused the latched data to be used by the TDM software. A minor modification was made to the test driver schema to simulate the way the multi-tasking executive would call the TDM entry point. This approach for calling the subsystem under test is commonly used with other TAF schemas (e.g., MATRIXx test driver schema) to propagate state data. This modification to the test driver schema produced a test driver that made two calls to the main entry point. The execution of the new test driver resulted in a failure that emulated the situation where state data would propagate and latch into a particular state. The test that uncovered the failure scenario is associated with the modeled requirement First_Marked_Bad, defined in Figure 4.

6. Summary

The TAF approach for automated model-based testing was applied to the Mars Polar Lander Touchdown Monitor software. The SCRtool was used to model textual requirements. The TAF toolset was used to translate the model, while T-VEC generated test vectors and test drivers. The TAF team traveled to the LMAO site and was able to use the toolset to identify an error that is the probable cause of the pre-mature shutdown of the Touchdown Lander thrusters. These results suggest that the TAF approach has the potential to provide more standardized and thorough testing for verification of critical software and system functionality.

6.1 Other applications and results

The core capabilities underlying this approach were developed in the late 1980s and proven through use in support of FAA certifications for flight critical avionics systems [BB96]. The approach supports requirement-based test coverage mandated by the FAA with significant life cycle cost savings [Sta99; Sta00; Sta01]. The approach reduces cost, effort, and cycle-time by eliminating requirement defects and automating testing [Saf00]. Safford's presentation summarized the benefits:

- Better quality requirements for design and implementation help eliminate rework in those phases as well as during test
- Verification modeling can reduce the time normally spent in verification test planning by up to 50 percent
- Test generation from a verification model can eliminate up to 90 percent of the manual test creation and debugging effort
- Both the number of test cases and the phasing of their execution can be optimized, eliminating test redundancy
- A known level of requirements coverage can be planned, and measured during test execution

The approach and tools have been used for modeling and testing software and systems. It has been applied to functional security testing [BBNC01], as well as, critical applications like telemetry communication for heart monitors, flight navigation, guidance, autopilot logic, display systems, flight management and control laws, airborne traffic and collision avoidance. The approach supports automated test driver generation open (e.g., C, C++, Java, Ada, Perl, PL/I, SQL) and proprietary languages.

7. Acknowledgements

This work has been a cooperative effort by various individuals within Lockheed Martin Aeronautics Company and the Software Productivity Consortium.

8. 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, 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, June 1997.

[BBNC01] Blackburn, M.R., R.D. Busser, A.M. Nauman, R. Chandramouli, Model-based Approach to Security Test Automation, In *Proceeding of Quality Week 2001*, June 2001.

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

[Sta99] Statezni, David, Industrial Application of Model-Based Testing, *16th International Conference and Exposition on Testing Computer Software*, June 14-18, 1999.

[Sta00] Statezni, David. Test Automation Framework, State-based and Signal Flow Examples, *Twelfth Annual Software Technology Conference,* April 30 - May 5, 2000.

[Sta01] Statezni, David. T-VEC's Test Vector Generation System, *Software Testing & Quality Engineering*, May/June 2001.

[Saf00] Safford, Ed L. Test Automation Framework, State-based and Signal Flow Examples, *Twelfth Annual Software Technology Conference*, April 30 - May 5, 2000.

[TVK90] Tsai, W.T., D. Volovik, T.F. Keefe, Automated test case generation for programs specified by relational algebra queries, *IEEE Transactions on Software Engineering*, 16(3):316-324, March 1990.

[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.