The Engineering of Model-Based Testing: Guidelines and Case Studies

Version 01.00.25
July 2005

Mark R. Blackburn, Ph.D.
Blackburn@knowledgebytes.net

Knowledge Bytes, LCC

CONTENTS

This page intentionally left blank.

FIGURES

This page intentionally left blank

TABLES

This page intentionally left blank

ACKNOWLEDGMENTS

The authors wish to thank

- Steve Allen, Mark B. Hall, Verlin Kelly, and Gerald Adams from Lockheed Martin
- Mike Polen from the Systems and Software Consortium, Inc.
- Chris Snyder, the key developer of the DOORS integration into TAF
- Roberta Troy, Documentation Consultant, for providing technical editing that enhanced the readability of this report

This work and the case studies described in this work were performed in conjunction with the Software Productivity Consortium (SPC) and Systems and Software Consortium (SSCI). At the closing of SSCI, the rights of this document were turned over to the author, Dr. Mark R. Blackburn.

This page intentionally left blank

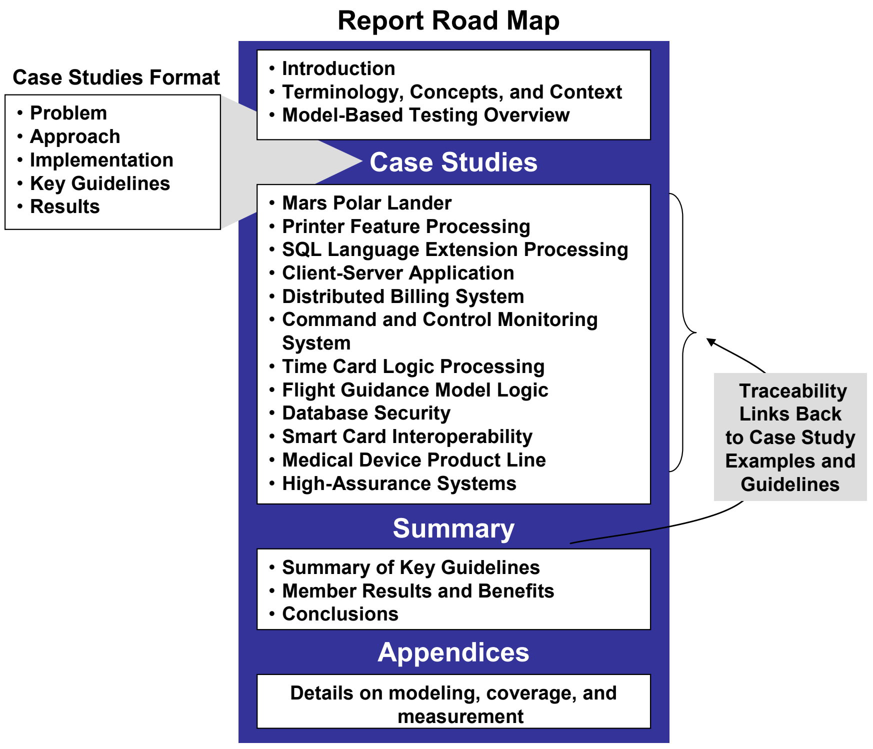
PREFACE

The purpose of this report is to explain the benefits, findings, and recommendations for adopting model-based testing. This report leads readers through a series of case studies that highlight member stories related to improving requirements, design, and systematic test, all fostered by the structure that model-based testing can impose on an organization to improve the ill-structured traditional testing processes.

The title of this report comes from reflecting on the experiences of a few members that have progressed small pockets of their organizations into the engineering of model-based testing. The report explains many of the hidden benefits related not only to more effective testing but also to improved requirements and design. The last case study in this report reflects on this mature approach to model-based testing that has been implemented in a few member companies.

How to Read This Report

This report is long, and readers should use this report as a reference by reading the case studies of interest based on the applicability of the problem and application area or the related guidelines that are covered in that case study.


Report Road Map
Case Studies Format
• Problem
• Approach
• Implementation
• Key Guidelines
• Results
• Introduction
• Terminology, Concepts, and Context
• Model-Based Testing Overview
Case Studies
• Mars Polar Lander
• Printer Feature Processing
• SQL Language Extension Processing
• Client-Server Application
• Distributed Billing System
• Command and Control Monitoring System
• Time Card Logic Processing
• Flight Guidance Model Logic
• Database Security
• Smart Card Interoperability
• Medical Device Product Line
• High-Assurance Systems
Traceability Links Back to Case Study Examples and Guidelines
Summary
• Summary of Key Guidelines
• Member Results and Benefits
• Conclusions
Appendices
Details on modeling, coverage, and measurement

The following list describes how different audiences can use this report:

- ***Senior Executives*** will want to read the Executive Summary and Section 15, the Summary, which provides return on investment (ROI) results claimed by Consortium members.
- ***Manager and Project Leads*** should read the Executive Summary and skim through Section 1.3, the Organization that give a general overview of each case study and the guidelines illustrated within each case study. The Summary section is useful, as it provides ROI, and a summary of the guidelines that are linked back to the case studies.
- ***New to model-based testing.*** Read Sections 2 and 3 to gain awareness of the terminology used in this document. Browse the case studies in Sections 4 through 14 to see how the same general model-based testing process is applied to a wide variety of applications.
- ***Some experience with modeling.*** Skim Sections 2 and 3 to gain awareness of the terminology used in this document. Browse the case studies in Sections 4 through 14 to see how the same general model-based testing process is applied to a wide variety of applications. Be sure to read the subsection titled Key Guidelines near the end of each case study to get specific guidance and pointers to more detailed information.
- ***Advanced perspective on organizing for multiteam, model-based test engineering.*** Section 14 pulls all the guidelines together and provides a historical perspective on evolving an organization into a multiproject, and multi-program engineering team that integrates model-based testing into the overall system and software development process.

EXECUTIVE SUMMARY

...even without any understanding of the system or requirements, the systematic process of the model-based test generation resulted in tests that were able to find the bug that is the likely cause of the Mars Polar Lander crash…paraphrase from Robert Knickerbocker – Member Forum

… the Collins lead for this pilot estimated that a Level A Flight critical project could save up to 52% of its total software development costs using a full model-based development environment, including auto-code and auto-test. [Consortium 2000]

Traditional testing is not working. Arguably, there are a few pockets of excellence, but to provide the quality and reliability of complex systems, testing must be engineered just like systems must be engineered. Poor quality starts with poor requirements. Model-based test engineering improves requirements to reduce program risk, while increasing the testability of the design, along the systematic testing of the implemented system.

The author has worked with members on many modeling applications in various application domains using requirement- and design-based modeling and test generation tools to reduce requirement defects and testing cost and effort, while improving quality and reliability. However, these applications and the benefits derived are not well-known. Managers and lead project engineers often are skeptical about trying modeling on their applications or systems.

Use of Case Studies

This report dispels many common technology adoption concerns by removing member proprietary information that has been captured since the pilot efforts in the late 1990s. The report provides examples and case studies for various applications, recommended approaches, and benefits derived from requirement modeling and model-based testing. These case study results should provide evidence for project managers and leads to justify a pilot project trial that will give them concrete evidence of the benefits and relevancy of this approach to their applications. The report blends documented examples, experiences, and lessons learned of past users with guidelines for requirement and interface modeling, supporting requirement defect identification and verification of different types of applications that will be beneficial to engineers.

Benefits to Member Companies

The audience for this report includes software and system engineers involved in requirements, design, and verification of software-intensive systems. This report integrates the concept of requirement modeling to support verification and validation, with test generation, and test driver generation to support the use of the models in the process of testing. The report provides guidance for a wide spectrum of application-based situations faced by developers and testers, which organizations can leverage in many different application areas. The report also discusses organizational best practices and provides the guidance in a prescription and example-driven manner.

The key benefits of the report include the following:

- ROI evidence for senior management and decision makers
- Evidence to managers and project leads that requirement modeling and model-based testing have been successfully applied on a wide range of application
- Better understanding of the possible applications of this technology and more specific data on the benefits that members have experienced
- Examples for line engineers that are more beneficial than generic training examples or simple tutorials
- Evidence that organizations that have applied the Test Automation Framework (TAF) are reducing cost and schedule through early identification and removal of requirement defects, better design for testability, better organization and allocation of requirements, and systematic automated testing
- Key best practice guidelines that are linked to case study examples

1. INTRODUCTION

This report is based on lessons learned by the Systems and Software Consortium, Inc. (SSCI) and members from deploying various types of modeling capabilities since 1996. This report presents information in the following manner:

- Uses a case studies format to discuss how organizations used specific tools to support requirement analysis, modeling, design for testability, and testing
- Discusses challenges, findings, recommendations, and best practices observed from the use of model-based testing tools
- Reflects on tool requirements that are essential for organizational adoption, including:
  - Support for requirement-to-test traceability from requirement management tools
  - Requirement and design modeling
  - Model-based test generation
  - Automated test execution and analysis
  - Test coverage analysis.

In addition, these model-based testing tools have qualification evidence to support use on safety-critical applications.

The integrated environment, generically referred to as the Test Automation Framework (TAF), integrates government and commercially available requirement and design modeling tools with test generation tools. TAF integrates the DOORS requirement management tool with the T-VEC Tabular Modeler (TTM) that supports the Software Cost Reduction (SCR) method [Alspaugh 1992] for requirement modeling. DOORS integrates also with Simulink, which supports design-based models, and TAF integrates requirement models with design models to provide full traceability from the requirements source to the generated tests, as shown in Figure 1. It integrates also with code coverage tools produced by LDRA and Rational/IBM as well as open source tools such as GNU.

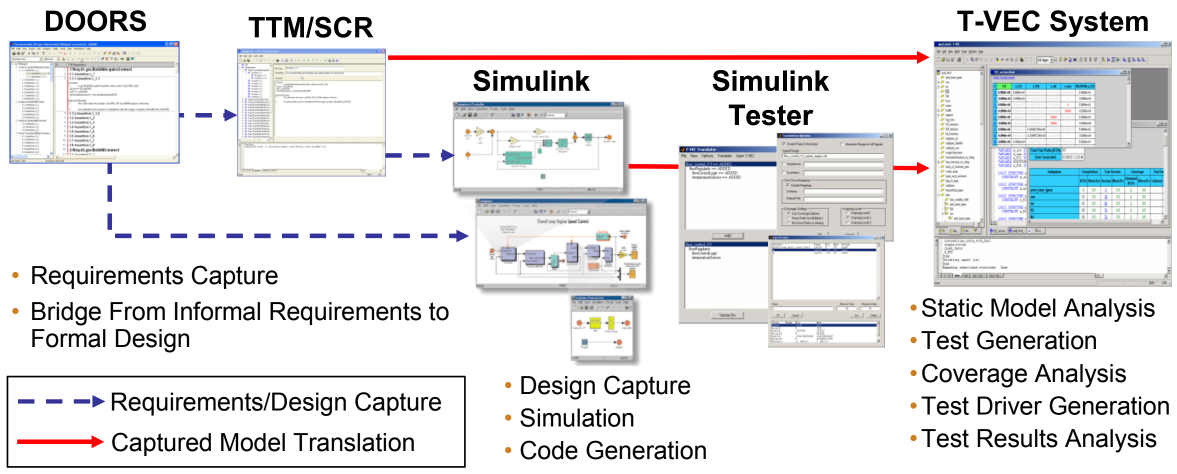


Figure 1. TAF Integrated Environment

1.1 SCOPE

This report integrates example applications with specific guidance for using an engineering-based approach to requirement and verification modeling to support requirement analysis and continuous automated testing based on the use of TAF and T-VEC. The report provides a series of modeling problems described as case studies. Each section provides guidelines that increase in complexity while covering various types of modeling situations that developers typically face.

This report addresses the relationship of the developed models to the associated tests and test environment. It describes approaches for developing the test drivers derived from models and explains the relationship between a hierarchical set of models and the associated tests that would be injected into implementations or simulations. It also discusses the concepts of model-based coverage as well as code coverage.

The report discusses some of the applications shown in Table 1 that cover software unit, integration, and system-level testing.[1] The models typically describe the functional requirements of a system or component, but one application describes security requirements for a database. The target implementations range from web-based to embedded systems on various platforms and operating systems (OS) and test drivers (aka test scripts) that were generated to support automated test execution in many languages and data formats.

Table 1. Application Summary

Application	Level	System/Component	OS	Test Language
Database Security	System	Oracle	Win2K, XP	Perl/Java/JDBC
Smart card interoperability	System	Reference implementation	Win2K, XP	Java
SQL extension language processing	System	Parallel Database	Win2K	SQL extension
Copier/printer feature processing	System	Custom hardware	Unix	XML
Client-server web application	System	Web-to-database	Win2K/IE	Winrunner
Client-server	System	Tracking and certification	Win2K/CISC	DynaComm
Distributed billing system	System	Custom application	Unix	Perl
Umanned vehicles	System	Brake control	Unknown	VB-like

[1] Because of time limitations, the authors could not cover the unmanned vehicle, sonar, and utility applications in this version of the case studies. For information on any of these applications, contact the author.

Application	Level	System/Component	OS	Test Language
Mars polar lander	Software unit	Touch down monitor	Win2K	C
Command control for ship	System	System monitor	UNIX	Slang script
Medical devices	Integration	Mode switch	UNIX/custom	C-like (custom)
Medical devices	Integration	Monitoring and method selection	Custom	Custom
Medical devices	Integration	Internal management	UNIX/custom	C-like (custom)
Flight guidance mode logic	Unit/integration	Mode logic	Custom	Java
Avionics monitoring	System	Cruise energy management	Custom	SWAT (custom)
Mission management	System	Stores management	Custom	SWAT (custom)
Sonar	Unit/integration	Mode control	Custom	HTML
Utility	Unit	Transfer time conversion	UNIX	C
Time card processing	System	Time card rules processing	Win2K, XP	Data file

1.2 AUDIENCE

The report is intended for program managers, project managers, project engineers, requirement engineers, designers, test engineers, technologists, quality assurance personnel, certification authorities, and line engineers that perform a variety of development and test functions. The benefits include the following:

- ***Program managers*** can use the report to understand the benefits derived from better understanding of the requirements with continuous testing to reduce the overall cost while achieving increased reliability and quality.

- ***Project Managers and Project Engineers*** can use the report to obtain a general awareness of the processes, models, and tools that are required to adopt model-based testing.

- ***Quality Assurance and Certification Authorities*** can use the report to help assess compliance with well-defined processes, standards, guidance on qualified tools, and related development and verification artifacts.

- ***Line Engineers*** should benefit from the examples, experiences, and lessons learned of past users that are documented throughout the case studies. The report provides guidelines and examples for requirement and interface modeling supporting requirement defect identification and verification of different types of applications that should be beneficial to line engineers.

- ***Test Engineers*** can use these guidelines to assist in selecting techniques and heuristics for developing verification and validation models that provide thorough requirements test coverage while also creating a detailed mapping between the requirement engineer's requirements information and the design/implementation engineer's design specifications.

- ***Requirement Engineers*** can use these guidelines to better understand how their work will be used for requirement analysis. The requirements engineer should learn how to use the results of such analysis to fill in important gaps in requirements information and correct defects and ambiguities in the requirements already specified. In addition, model-based analysis supports validation of the requirements to better ensure that the final product meets the custom need.

- ***Designers/Implementers*** can use these guidelines to better understand the importance of designing a system for testability and how their work is used to support early development of test drivers and how the application of these test drivers can be used to complement the debugging process. In addition, the design/implementation engineer should learn how to use test failure information continuously during the development process to enhance the final product. The design engineer should better understand the importance of identifying and specifying complete information for the system interfaces, including data value ranges and type representation.

1.3 ORGANIZATION

The report is structured as follows:

- ***Introduction***. Section 1 provides a general introduction to the scope of the report, an overview of the case studies, and guidelines for reading this report.
- ***Terminology, Concepts, and Context.*** Section 2 provides some definitions, introduces modeling concepts and tools, and provides a summary of the TAF capabilities, including a recommendation for its use by the United Kingdom Ministry of Defence (UK MoD) for use on high-assurance systems. This section briefly discusses TAF's design modeling support and relationship to other tools.
- ***Model-Based Testing Overview.*** Section 3 provides introduces the general process for requirement-based modeling and automatic test generation. Each case study and nearly all TAF usage follow this general pattern.
- ***Mars Polar Lander.*** Section 4 is a case study on the Mars Polar Lander (MPL) application. It provides a comprehensive, but high-level description of the model-based testing process. The key guidelines include:
  - Goal-driven modeling, modeling threads of an application as opposed to the entire application
  - Specifying the negative cases of a requirement
  - Design of pattern for test driver generation for applications with state data
- ***Printer Feature Processing.*** Section 5 describes a case study for modeling the Printing Instructions Format Specification (XPIF) of the Internet Printing Protocol (IPP) and associated embedded application testing. The key guidelines include:
  - Importance of understanding the interfaces and test environment before modeling
  - Selecting an application feature for a pilot project
  - Guidelines for selecting an initial projects
- ***SQL Extension Language Processing.*** Section 6 is a case study that describes the modeling for a language extension to the Standard Query Language (SQL) for a parallel database management system. The key guidelines include:
  - Modeling a programming, scripting, or command language grammar associated and the generated tests that cover all the combinations of the possible language statements
  - Constructing language commands during test driver generation

- ***Client-Server Web Application.*** Section 7 provides a case study for modeling a web-based application that integrates with a database server. It describes how test drivers are produced to use the WinRunner tool to support the GUI-based test execution. The key guidelines include:
  - An effective technology adoption process
  - Use of the WinRunner application program interface (API) functions to support automated test execution
- ***Distributed Billing System.*** Section 8 describes the case study for a component of a distributed billing system. The key guidelines include:
  - How to produce models and generate test cases in one environment that execute in a different target environment
  - Test drivers that dynamically generate test data in real time
  - Embedded test driver that monitors and logs the outcomes of the test driver executing in the target environment to be used in the test results analysis
- ***Command and Control Monitoring System.*** Section 9 is a case study associated with a monitoring component for an onboard command and control system, and the estimated ROI. It summarizes key guidelines:
  - Effective technology adoption approach combined with pilot project and tailored training
  - Using a simulator for test execution
  - How to use the modeling to drive improvements in the design of the target system and simulators
  - Leveraging an embedded data recorder to capture the actual outputs of a tests
- ***Time Card Logic Processing.*** Section 10 is a case study that describes the modeling of business rules of a component of an information technology (IT) company. This case study illustrates a good first application for an organization that wants to learn how to do test automation that also has limited documentation of the requirements. This section uses this simple example to summarize key guidelines:
  - Modeling practices such as naming conventions, the use of constants, and traceability links
  - Terms that can be reused throughout the model
  - Test vector generation from hierarchical models
  - Model defects
  - Traceability to requirements
  - Test generation from inlined models
- ***Flight Guidance Model Logic.*** Section 11 is a case study on one of the early applications of TAF applied to the Flight Guidance Mode Logic of an avionics system. This case study documents the finding by a member company and describes the importance of applying requirement-based modeling and testing. It discusses how the tools helped uncover 52 of 85 defects that were not identified using manual inspection processes. The key guideline discusses the judicious use of modeling with event specification.

- ***Database Security.*** Section 12 is a case study that describes the modeling of functional security requirements for the Common Criteria of the Oracle8 Database Server. The key guidelines include:
  - Modeling the dynamic generation of database content that avoids the costly effort of developing and maintaining a "gold" database
  - Modeling the positive as well as negative cases that represent potential security violations
  - Analysis of the interface and requirement dependencies
  - Design of model and test driver object mappings that provide reuse to reduce cost and maintenance
- ***Smart Card Interoperability.*** Section 13 is a case study that describes the modeling for the Government Smart Card Interoperability Specification (GSC-IS) of the National Institute of Standards and Technology (NIST). The key guidelines include:
  - Importance of test driver generation and test infrastructure middleware to support test execution and logging
  - Modeling of dependencies and the relationship to test sequencing
  - Uses of requirement traceability
- ***Medical Device Product Line.*** Section 14 is a case study describes the multiphased technology adoption by a company that produces product families of life-critical medical devices. The key guidelines include:
  - Improved requirements process
  - Improved test infrastructure
  - Design for testability
  - Modular requirement specifications
  - Organizational adoption process
  - Multiteam model and test infrastructure organization
  - Modelbased review practices
  - Cost benefits of modeling requirement early
  - Model-based measurement for project management
  - Configuration management of model-based artifacts
- ***Summary.*** Section 15 provides a summary and conclusions.
- ***SCR Requirement Modeling.*** Appendix 0 provides a brief overview of the SCR modeling method and concepts.
- ***Code Coverage and Structure Testing.*** Appendix 0 discusses the difference between code coverage, structural testing, and model coverage.
- ***Measurement Information Product and Measurement Construct.*** Appendix 0 provides measurement concepts and information underlying TAF measurement.

1.4 HOW TO READ THIS DOCUMENT

The following list describes how to get the most out this document, based on the reader's level of modeling experience:

- ***New to model-based testing.*** Read Sections 2 and 3 to gain awareness of the terminology used in this document. Browse the case studies in Sections 4 through 14 to see how the same general model-based testing process is applied to a wide variety of applications.
- ***Some experience with modeling.*** Skim Sections 2 and 3 to gain awareness of the terminology used in this document. Browse the case studies in Sections 4 through 14 to see how the same general model-based testing process is applied to a wide variety of applications. Be sure to read the subsection titled Key Guidelines near the end of each case study to get specific guidance and pointers to more detailed information.
- ***Advanced perspective on organizing for multiteam, model-based test engineering***. Section 14 pulls all the guidelines together and provides a historical perspective on evolving an organization into a multiproject and multiprogram engineering team that integrates model-based testing into the overall system and software development process.

1.5 RELATED DOCUMENTS

The interaction with members using TAF has helped spawn other reports related to this capability, such as reliability, measurement, subcontract compliance sign-off, and safety-critical systems. Following is a list of related products:

1. *Objective Measures for V&V and Software Reliability*, White Paper [SSCI 2005]
2. *Guidance for Achieving Mission Assurance in Software-Intensive Systems*, Technical Report [Consortium 2004a]
3. *Model-Based Verification and Validation for Security Requirements of Systems*, Technical Report [Consortium 2004b]
4. *Requirement-Based Verification Sign-Off for Subcontract Integration Compliance*, Technical Report [Consortium 2004c]
5. *Automatic Code Generation: State of the Practice*, SPC Technical Report [Consortium 2004d]
6. *Strategies for Web and GUI Testing*, Technical Report [Consortium 2004e]
7. *Model-Based Development and Automated Testing*, Course [Consortium 2003a]
8. *Testing Complex Systems*, Course [Consortium 2003b]
9. *Guidelines for Software Tool Qualification*, Technical Report [Consortium 2003c]
10. *Guidelines for Using Test Automation Framework Measures*, Technical Report [Consortium 2003d]
11. *Test Automation Framework for T-VEC and Simulink*, Course [Consortium 2003e]
12. *Mars Polar Lander Fault Identification Using Model-based Testing* [Blackburn 2001]
13. *Applying the Test Automation Framework With Use Cases and the Unified Modeling Language*, Technical Report [Consortium 2002]

14. *Specification Transformation to Support Automated Testing*, Technical Report [Consortium 1998]
15. *Test Automation Framework*, [Consortium 1997]

1.6 TYPOGRAPHIC CONVENTIONS

This report uses the following typographic conventions:

Serif font	General presentation of information
Italicized serif font	Publication titles and names of words or expressions used in grammar rules
Boldfaced italicized serif font	Run-in headings in bulleted lists
Boldfaced serif font	Section headings and emphasis
`Courier New` font	Algorithms or code fragments

2. TERMINOLOGY, CONCEPTS, AND CONTEXT

This section does the following:

- Defines terms and concepts
- Provides some background information to set the context for the report and some general process information for using requirement-driven modeling
- Describes the basic elements of a model and presents primary modeling concepts and related foundational principles.

2.1 TERMINOLOGY

Terminology can be a source of confusion; therefore, this section provides definitions for terms used in this report. Some definitions refer to requirement concepts, while others refer to elements of the target system that is designed or implemented.

- ***Architecture.*** Specification of component relationships and their input and output interfaces.
- ***Component.*** Software item for which a separate specification is available.
- ***Functional Design Specification.*** Captured description of the functional design.
- ***Functional Design.*** How a functional requirement specification is satisfied.
- ***Functional Requirement Specification.*** Description of the input-to-output relationships of a component with respect to its interfaces within the environment.
- ***Functional Requirement.*** Captures the nature of the interaction between a component and its environment.
- ***Implementation.*** How a functional design specification and nonfunctional requirements are satisfied.
- ***Model.*** (1) *See* Section 2.3; (2) sometimes used in a less formal manner to refer to a structure of elements and the heuristics for relating the elements.
- ***Nonfunctional Requirements.*** The “ilities” (e.g., reliability, availability, maintainability, testability, enhanceability) that result in constraints on the design that manifest themselves in the implementation.

- ***Requirement Thread.*** Functional requirement for a component output.
- ***Specification.*** Used generically to refer to functional requirement, functional design, or test specifications; sometimes the word model is used synonymously with the word specification.
- ***Test Vector.*** Includes test inputs, test input values, expected outputs, expected output values, and a mapping of each test to the associated specification element.
- ***Thread.*** Execution sequence; can span components and architectural levels.
- ***Validation Model.*** A refinement of a product focused on supporting test automation. A validation model represents the correct, complete, consistent, and testable requirements but describes functional behavior in terms of the architecture that is represented by the requirements.[2]
- ***Validation.*** Evidence that the right product is defined in the requirements.[3]
- ***Verification Model.*** A refinement of a requirement focused on supporting test automation. A verification model represents the requirements but describes functional behavior in terms of the interfaces that are represented by a design and associated implementation.
- ***Verification.*** Evidence that the product was built to the requirements defined.[4]

2.2 COMPONENT

SSCI members typically produce large and complex components that are composed of other components (subcomponents). Figure 2 depicts the conceptual context for integrating a component into an overall system architecture. The term component is used generically to refer to an element of a larger system. A component can be software only or some combination of software and hardware. The specification of the component defines the interfaces and the behavior of some Component C. The integration of a component must be correct syntactically, which means that the interfaces to exchange information (e.g., messages, parameters, data structures) are syntactically interpreted consistently by Component C and Architecture A. A common integration problem is related to the inconsistent interpretation of the behavioral requirements associated with the semantics of processing and producing the information of Component C (e.g., the number representing distance produced by C is in feet but should be in meters). The model-based analysis and testing case studies discussed in this report address defining interface and behavioral requirements of a component.

[2] Supplied and requested for inclusion by Lockheed Martin.
[3] Supplied and requested for inclusion by Lockheed Martin.
[4] Supplied and requested for inclusion by Lockheed Martin.

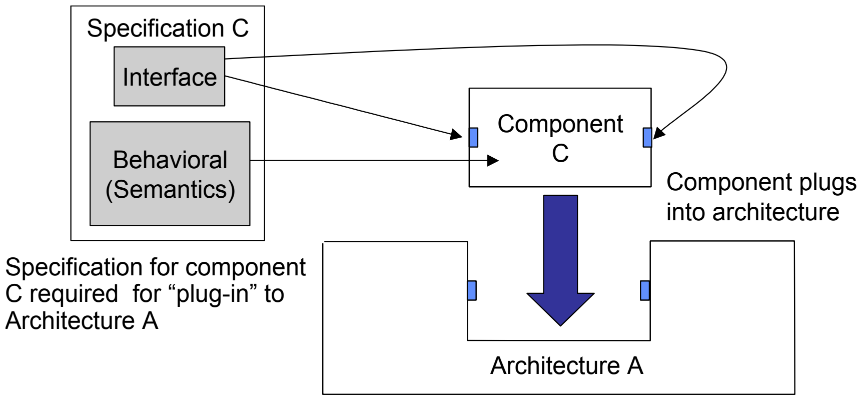

Specification C
Interface
Behavioral (Semantics)
Component C
Component plugs into architecture
Specification for component C required for "plug-in" to Architecture A
Architecture A

Figure 2. Conceptual Framework for Component Integration

A component can include other components as shown in Figure 3. Subcomponents such as C.1 through C.5 may support large sets of high-level functions at an enterprise level or low-level functions such as software packages for communication processing, data processing, operating system control, or data storage. Therefore, the specification of Component C must be refined by the designer or architect into a number of lower-level specifications such as C.1 through C.5. The model-based analysis and testing also addresses component integration and verification.

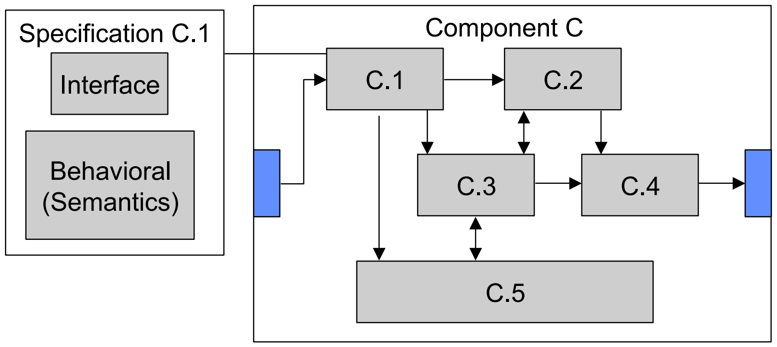


Figure 3. Concept of Subcomponents and Component Elements

2.3 MODELING CONCEPTS

Engineers use different types of models everyday. Models are used to help manage complexity when describing aspects of a system within different system contexts; this is sometimes referred to as abstraction.

For example, the source code of a program is a model. A compiler transforms it into a form that can be loaded into the target machine to fully enact the essence of the model. Any program developer knows that it takes rules to construct a working system from a model defined as source code statements.

The most obvious use of a model is to characterize a program's function using requirement and design specifications. Using artifacts captured within a model, developers can use processes such as verification—including testing, analysis, and reviews—to assess the compliance of a program with its specifications. A model must have the following characteristics:

- A modeling language
- Rules for using the model, typically referred to as a method
- A structure for organizing and relating the artifacts of the model

2.4 MODELING CONTEXT

The TAF/T-VEC method and tools were designed originally to support the verification and validation of high-assurance applications. TAF/T-VEC has been used in Federal Aviation Administration (FAA) and Food and Drug Administration (FDA) projects, and several members are planning to use TAF/T-VEC on programs that require certification or sign-off by these agencies. In addition, the UK MoD has assessed it and recommends its use. Some of the unique characteristics of TAF/T-VEC that are discussed in this section include the following:

- Supports both validation and verification
- Provides tool qualification for FAA/DO-178B [and the FDA
- Has been applied to safety-critical applications dating back to Traffic and Collision Avoidance System (TCAS) certification in 1990
- Identifies model defects early to reduce expensive rework
- Can be used to prove safety properties about models
- Has been applied to security threat modeling
- Generates tests from requirement or design models
- Verifies full test coverage with at least one test per requirement
- Works on any platform
- Provides full requirement-to-test traceability
- Generates measurement and status reports
- Integrates with modeling, requirement management, and code coverage tools

The TAF method for requirement and design-based model and verification was defined to meet the desired dependability requirements through the use of a constructive approach. This method, when applied with supporting tools, provides significant support for preventing and removing faults. The method provides the engineering rigor for allowing systematic specification analysis and automated specification-based testing. In addition, with appropriate metrics collection and metrics models, the resulting test data provides significant information to forecast system dependability parameters.

The TAF method is an engineering approach that uses related models for capturing architectural, requirement, design, implementation, and test specifications that are fundamental to the specification, verification, documentation, and implementation phases of software development. The method provides rules for managing requirement complexity through hierarchical and inheritance relationships of reusable specifications. The underlying processes help flush out requirement problems during the early stages of the project development.

TAF integrates various model development and test generation tools to support defect prevention and automated testing of systems and software. Although other modeling tools have been developed for TAF, Figure 4 shows only those component elements that have tool qualification support.

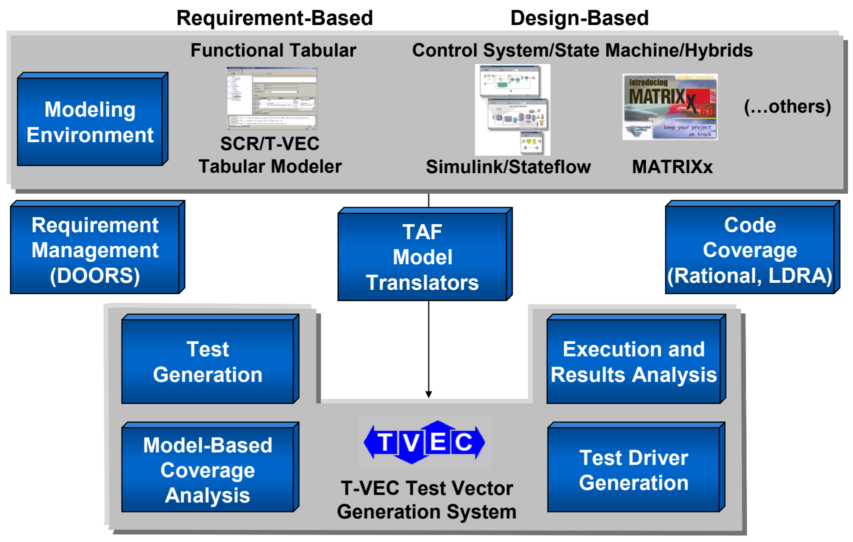

Requirement-Based
Design-Based
Functional Tabular
Control System/State Machine/Hybrids
Modeling Environment
SCR/T-VEC Tabular Modeler
Simulink/Stateflow
MATRIXx
(…others)
Requirement Management (DOORS)
TAF Model Translators
Code Coverage (Rational, LDRA)
Test Generation
Execution and Results Analysis
Model-Based Coverage Analysis
TVEC
T-VEC Test Vector Generation System
Test Driver Generation

Figure 4. TAF-Integrated Components

The case studies primarily discuss TAF's support for model analysis and test generation for requirement-based tools. However, TAF supports model analysis and test generation for design-based modeling, simulation, and code-generation tools such as MATRIXx and MathWorks' Simulink and Stateflow tools. Each of these tools integrates with T-VEC through a translator, which transforms each respective model into a form suitable for processing by T-VEC. Once a model is translated, users can generate tests using T-VEC through a graphical user interface (GUI) or command-line interface. TAF also integrates with requirement management tools such as DOORS. This integration allows requirements to be traced through the models to the test vectors and test drivers. When a failure occurs, the source of the failure can be traced back through the vectors, to the model, and to the requirements. Also, TAF integrates with different test-code coverage tools (e.g., Rational Test Realtime and LDRA's Testbed); see Appendix 0 for more details.

For design-based modeling approaches, the process tends to resemble the illustration in Figure 5. Simulink/Stateflow and MATRIXx are hybrid, control system modeling and code generation tools. In this scenario, models undergo translation and static analysis to verify their integrity. Model problems are reported by the T-VEC tools to the engineer responsible for constructing the model for immediate correction. Once modeling is complete, the model is used as the basis for developing tests. Through dynamic analysis (i.e., execution) of the system, anomalies in the model and implementation can be identified and corrected.

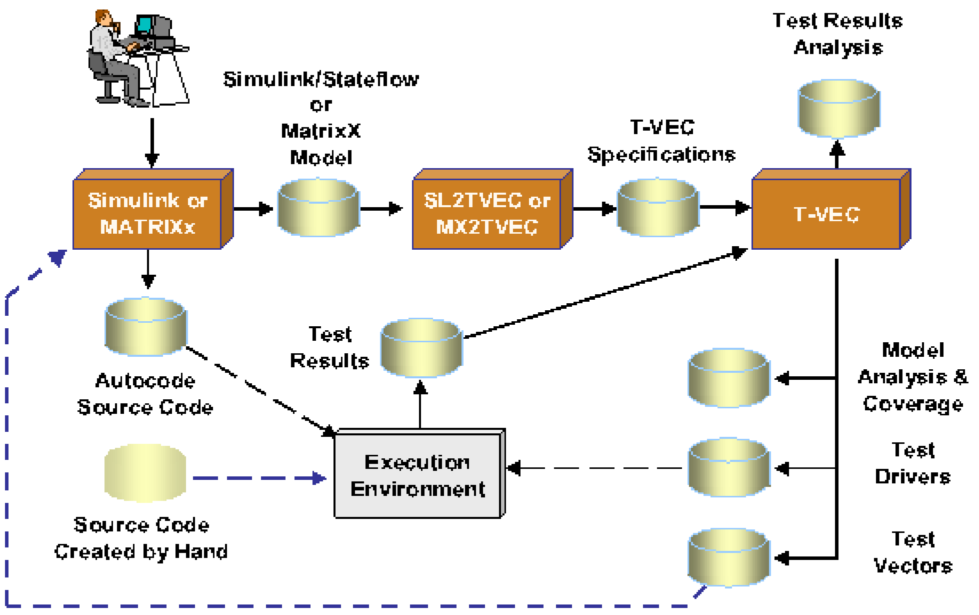


Figure 5. Simulink/Stateflow and MATRIXx Modeling Process Flow

2.5 REQUIREMENT-BASED MODELING METHOD AND TOOL

The case studies in this report are based on the modeling approach referred to as the SCR method and a tool called TTM that extends the SCR method. This section discusses a few details about the SCR modeling concepts and rules for using SCR for requirement-based automated testing. Images of TTM model elements are used throughout the case studies. This section introduces the TTM tool and briefly describes a few features that extend the SCR method. Appendix 0 provides additional, more fundamental details about the SCR method. For other specific guidance, the TTM tool has embedded help and a user's guide that provides details about the SCR method, along with tutorials to lead a user in developing models.

As shown in Figure 6, there are nine model element classes in TTM:

1. Info
2. Requirements
3. Types
4. Constants
5. Inputs
6. Assertions
7. Mode Machines
8. Terms
9. Outputs

The Types, Constants, and Inputs are related directly to the interfaces that must be defined for a component. Functional behavior is specified using combinations of Assertions, Mode Machines, Terms, and Outputs. The model element Info supports user-defined information; for example one member uses it to manage the configuration control version of the model. The Requirements element is a feature discussed in greater detail in Section 2.5.2.

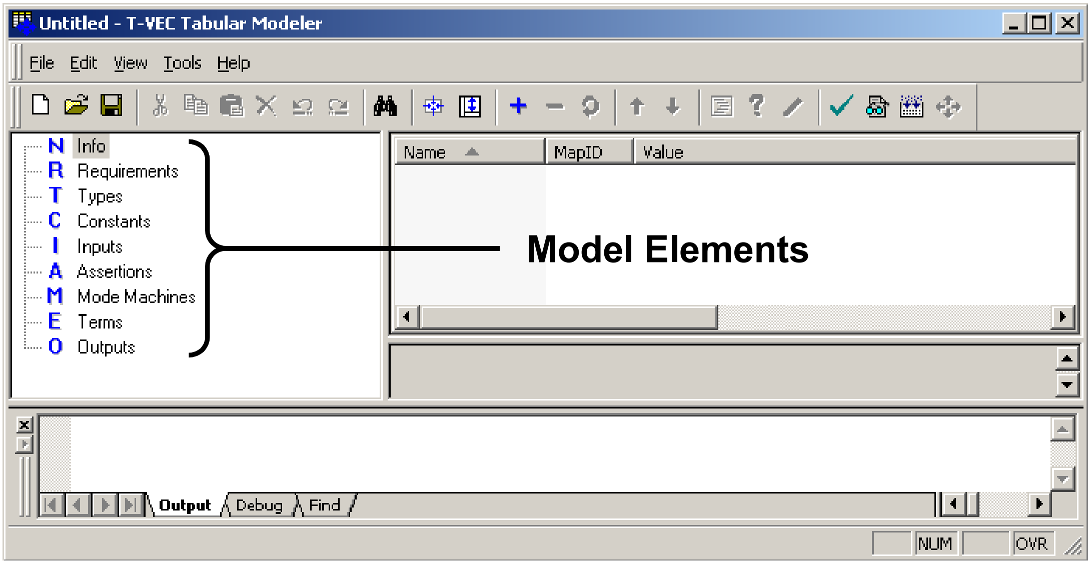
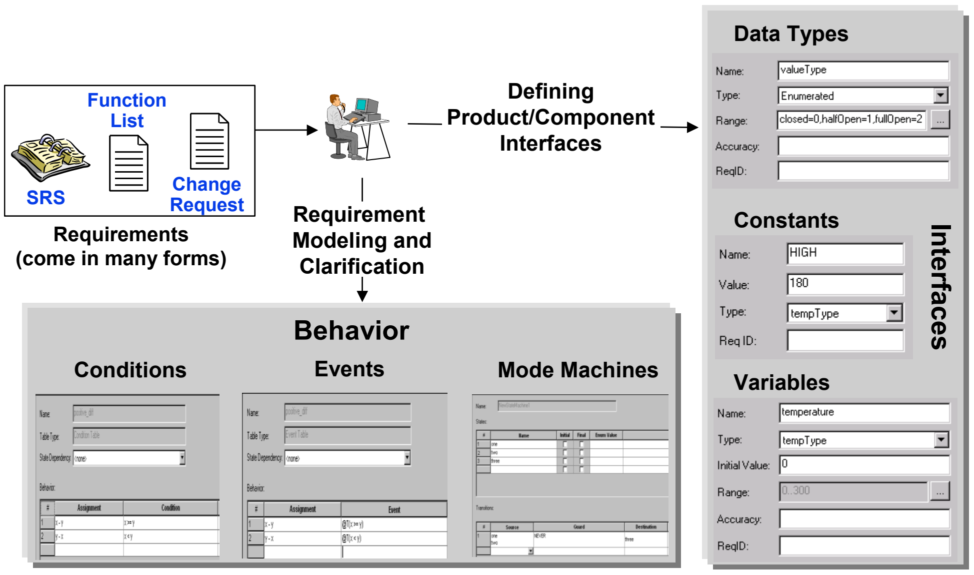


Figure 6. TTM Model Elements

2.5.1 Modeling Constructs

Simplistically stated, the SCR method is based on the use of decision tables and state machines to describe required behavior of some component. The structure for organizing and relating SCR model elements is based on tables, as shown in Figure 7. Tables are used to define data types and variables of the problem. Variables can be defined in terms of primitive types (e.g., Integers, Float, Boolean, Enumeration), or user-defined types. The behavior is modeled using combinations of tables that define functional aspects of the problem using a form of state machines (called Mode Tables), Condition, or Event Tables.


Function List
SRS
Change Request
Requirements (come in many forms)
Requirement Modeling and Clarification
Defining Product/Component Interfaces
Behavior
Conditions
Events
Mode Machines
Data Types
Constants
Variables
Interfaces

Name:	valueType
Type:	Enumerated
Range:	closed=0,halfOpen=1,fullOpen=2
Accuracy:	
ReqID:	

Name:	HIGH
Value:	180
Type:	tempType
Req ID:	

Name:	temperature
Type:	tempType
Initial Value:	0
Range:	0..300
Accuracy:	
ReqID:	

Figure 7. SCR/TTM Modeling Elements

The development of a model relates what system components have to do and how they have to do it. Then, through the generated tests, the model provides a measure of how well a target implementation satisfies the modeled requirements. The elements of "how" a system has to do its function is defined in terms of a set of interfaces specified with model variables and their associated data types. From a high-level perspective, models specify behavior relating input variables to output variables. Models also can represent behavior in terms of historical variables that are referred to as modes or terms.

2.5.2 Modeling Extensions

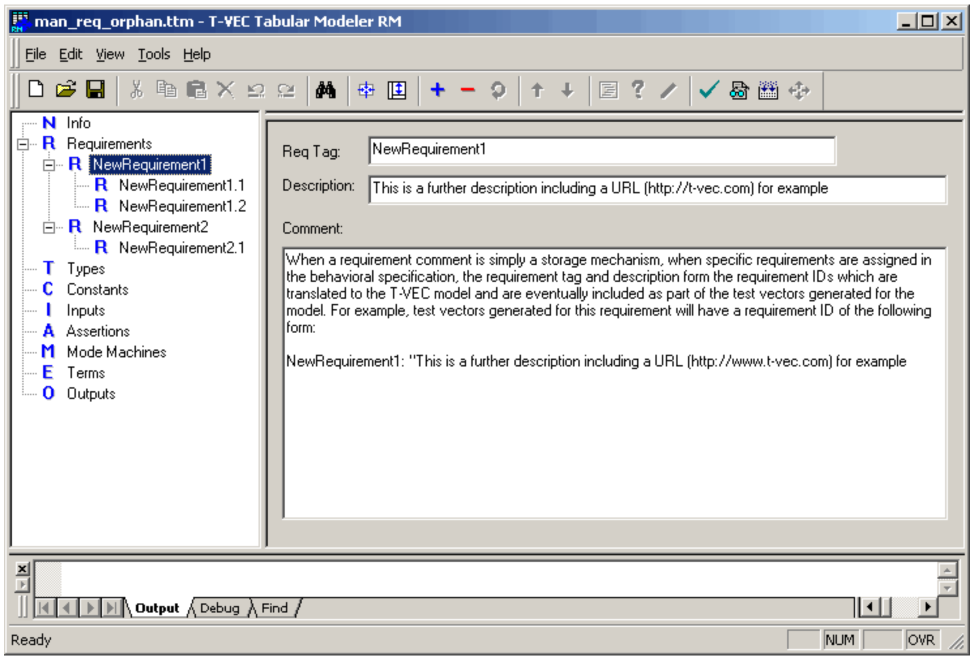
Member company use has guided extensions to the TTM tool, and SSCI plans several more. This section discusses two extensions: requirements and model includes.

2.5.2.1 Requirements Management

TTM manages requirements through a hierarchical decomposition (i.e., outline format) where each requirement is composed of the following:

- ***Tag.*** A unique identifier for the requirement comprised of letters, numbers, underscores and periods.
- **Description.** A single line of text further describing the requirement.
- ***Comment.*** Any additional text.

The hierarchy of requirements is managed through the model view, and requirements are decomposed by creating child requirements that display below their parents within the model view, as shown in Figure 8. Requirements are then linked to the model as shown in Section 2.5.2.2.



Req Tag: NewRequirement1

Description: This is a further description including a URL (http://t-vec.com) for example

Comment:

When a requirement comment is simply a storage mechanism, when specific requirements are assigned in the behavioral specification, the requirement tag and description form the requirement IDs which are translated to the T-VEC model and are eventually included as part of the test vectors generated for the model. For example, test vectors generated for this requirement will have a requirement ID of the following form:

NewRequirement1: "This is a further description including a URL (http://www.t-vec.com) for example

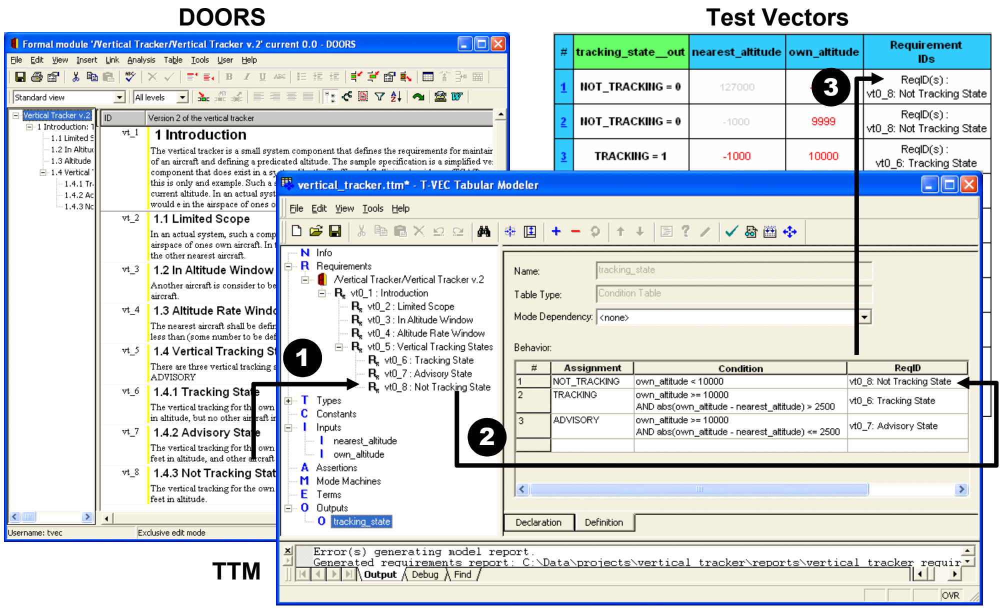
Figure 8. Hierarchical Requirements Example

2.5.2.2 Requirement-to-Test Traceability

This section provides an example to explain the process for linking DOORS requirements to the TTM requirements model. The tool support for requirement-to-test traceability involves linking various sources of requirements through the model. The model transformation, test vector generation, and test driver generation provide the tool support to link the requirements to the test vectors, test drivers, and test reports. The process, as shown in Figure 9, has three basic steps:

1. A DOORS module is imported into the TTM. There are options to add or delete a DOORS module to TTM or synchronize DOORS modules when they are updated. There is a one-to-one correspondence between a DOORS ID and a TTM requirement ID.
2. Imported requirements maintain the outline structure that they have within the DOORS environment. One or more DOORS requirements can be linked to an element of a TTM model (e.g., condition/assignment) as shown in Figure 9, or linked to a higher level in the TTM model, such as a condition, event, or mode table as shown in Figure 10.
3. The model translation maintains the link between the requirement ID, and during test generation, the requirement link is an attribute of the test vector. During test driver generation, requirement IDs can be output to the test driver to provide detailed traceability to the executable test cases.

TTM provides requirement management functionality that is similar to a DOORS module. Imported DOORS modules are linked into TTM as read-only modules. Changes to the requirements must be made within DOORS and then synchronized within TTM. Additional requirements can be created directly in TTM if they are not contained within DOORS or if the source requirements are not in a requirement management system such as DOORS. The process to link a requirement to the model is the same.



#	tracking_state__out	nearest_altitude	own_altitude	Requirement IDs
1	NOT_TRACKING = 0	127000	[illegible]	ReqID(s) : vt0_8: Not Tracking State
2	NOT_TRACKING = 0	-1000	9999	ReqID(s) : vt0_8: Not Tracking State
3	TRACKING = 1	-1000	10000	ReqID(s) : vt0_6: Tracking State

Name: tracking_state

Table Type: Condition Table

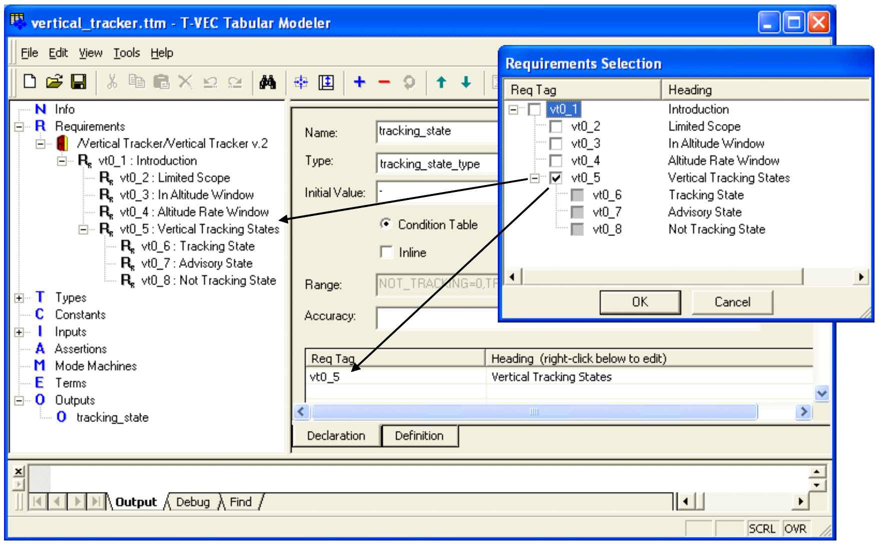
Mode Dependency: <none>

Behavior:

#	Assignment	Condition	ReqID
1	NOT_TRACKING	own_altitude < 10000	vt0_8: Not Tracking State
2	TRACKING	own_altitude >= 10000 AND abs(own_altitude - nearest_altitude) > 2500	vt0_6: Tracking State
3	ADVISORY	own_altitude >= 10000 AND abs(own_altitude - nearest_altitude) <= 2500	vt0_7: Advisory State

```
Error(s) generating model report.
Generated requirements report: C:\Data\projects\vertical_tracker\reports\vertical_tracker_requir
```

Figure 9. Requirement Links From Model to Test Vectors



N Info
R Requirements
/Vertical Tracker/Vertical Tracker v.2
vt0_1 : Introduction
vt0_2 : Limited Scope
vt0_3 : In Altitude Window
vt0_4 : Altitude Rate Window
vt0_5 : Vertical Tracking States
vt0_6 : Tracking State
vt0_7 : Advisory State
vt0_8 : Not Tracking State
T Types
C Constants
I Inputs
A Assertions
M Mode Machines
E Terms
O Outputs
O tracking_state

Name: tracking_state
Type: tracking_state_type
Initial Value: -
(● Condition Table) ☐ Inline
Range: NOT_TRACKING=0,TR
Accuracy:

Req Tag	Heading (right-click below to edit)
vt0_5	Vertical Tracking States

Req Tag	Heading
☐ vt0_1	Introduction
☐ vt0_2	Limited Scope
☐ vt0_3	In Altitude Window
☐ vt0_4	Altitude Rate Window
☑ vt0_5	Vertical Tracking States
▨ vt0_6	Tracking State
▨ vt0_7	Advisory State
▨ vt0_8	Not Tracking State

Figure 10. Linking Requirements to Table

2.5.2.3 Model Includes

TTM models support the inclusion of existing models of other requirements, interfaces, or functional behavior. As a result, this feature helps consolidate behavior common to multiple models into a single model and includes it in other models where needed. This feature also supports partitioning a model to allow multiple engineers to work on it in parallel. Section 14.4 discusses key guidelines.

3. MODEL-BASED TESTING OVERVIEW

This section describes the typical scenario for using TAF to support requirement-based modeling and automatic test generation. Any member of a team can develop models using requirement and interface information that is available and pertinent to that component of the system.

3.1 TRADITIONAL VERIFICATION AND VALIDATION

The traditional Validation and Verification (V&V) process often is discussed in terms of the V-model shown in Figure 11. There are various definitions for V&V, but [Boehm 1984] defined a simple working definition, where Validation is "building the right system," and Verification is "building the system right." Verification focuses on ensuring that the implementation satisfies the requirements. It is important to do early requirements validation to ensure that the system meets the user's needs and the stated requirements are correct, complete, and consistent. However, often the customer requirements are stated at such a high level that it is the responsibility of the system developer to work with the customer to decompose the high-level requirements into lower-level, more concrete, and testable requirements and design specifications. For example, "The system must be designed for fail-safe operation" is a nonfunctional requirement that manifests in many implementation-derived functional requirements. The developer must verify this requirement because it is allocated to components of the entire system that must implement the fail-safe design (e.g., redundant computers). In complex systems, this can be challenging.

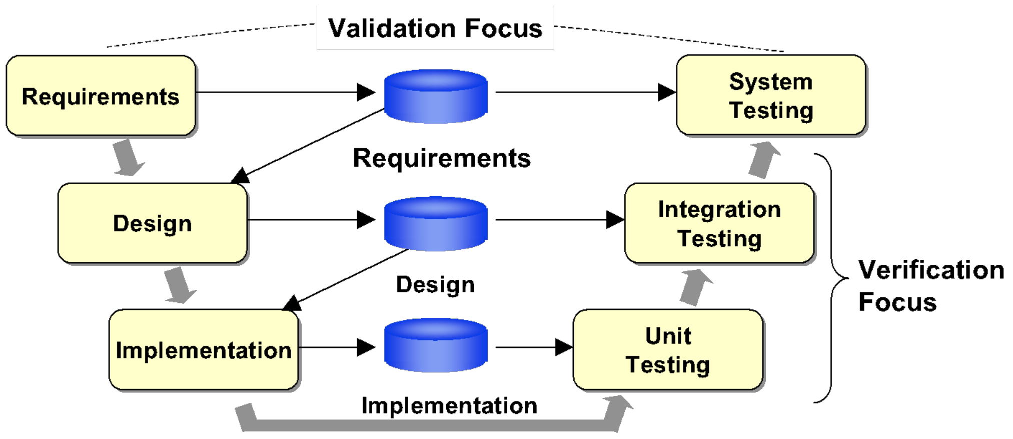


Figure 11. Typical V-Model

At any level of a system, sets of components represent the target system that must be represented in an architecture that characterizes its operational environment. For example, many of SSCI's members are suppliers to the Joint Strike Fighter program. The prime contractor specifies the overall avionics architecture, possibly in collaboration with the subcontractor (partners), and this architecture provides formalized interfaces for other components (e.g., radar, altimeters, guidance, electrical power, engines, and weapons). The architecture that defines the interfaces of these components defines the context for the component under development. Therefore, in terms of the interfaces specified at the high level, requirements allocated to the components must be specified in terms of the interfaces. Those requirements are analyzed, prioritized, and decomposed, and then tests are defined in terms of those allocated requirements.

The design decisions manifest in a lower-level set of components, and this process can be repeated recursively for each lower-level component as reflected in Figure 12. The initial customer requirements can be specified in any form from customer scenarios, use case, or safety cases, and they should be considered at the system level and traced down through the various levels of system refinement. Although Figure 12 shows decomposition into software levels, the system decomposition includes hardware too.

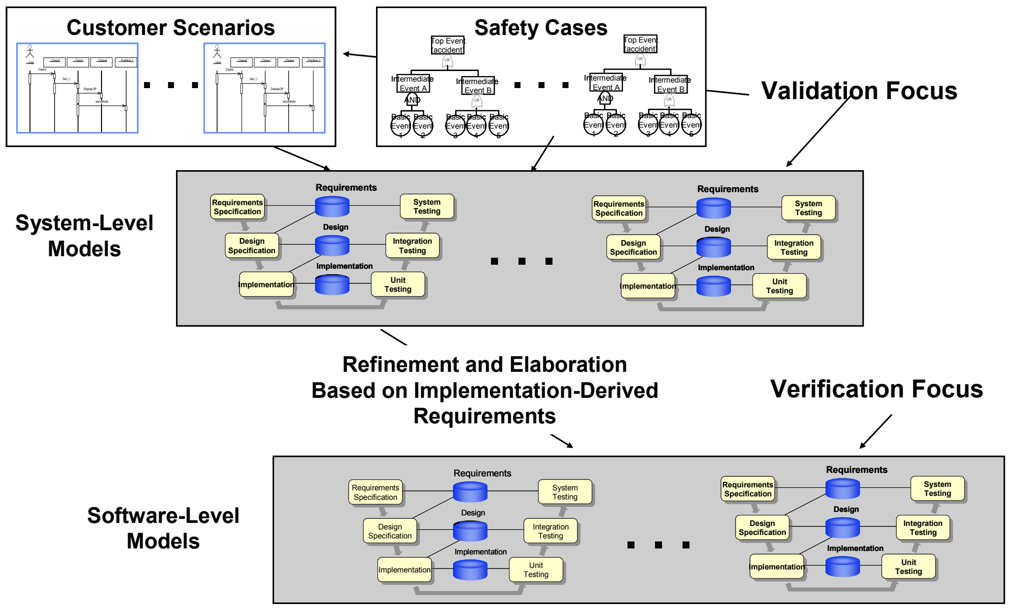


Figure 12. Hierarchical System Levels

3.2 TYPES OF TESTING

These hierarchical system levels support different levels of verification, validation, and testing. As discussed in Section 2.2, the component of a system can be a low-level software unit that is unit tested, a higher-level piece of software that is integration tested, or a packaged system that is system tested. There are often different types of testing performed by different people within the organization and by other stakeholders. Unit testing often is performed by the designer/implementer. Developers and test engineers often perform integration testing, and test engineers, certification organizations, and customers may do system testing. However, the same basic TAF process is applied to the development of a model to support testing for these different types and levels of testing as reflected in Figure 13.

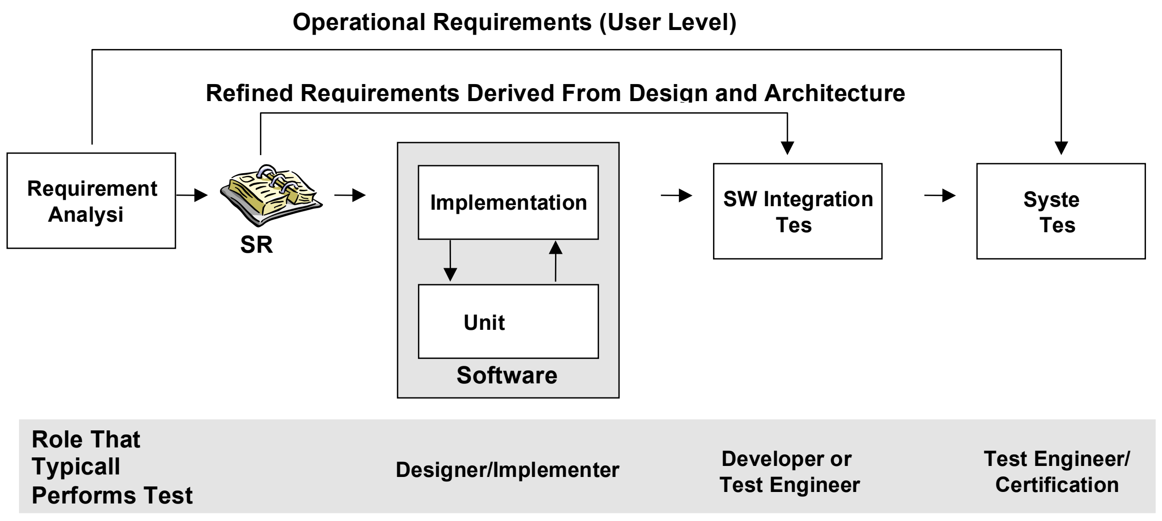

Operational Requirements (User Level)
Refined Requirements Derived From Design and Architecture
Requirement Analysi
SR
Implementation
Unit
Software
SW Integration Tes
Syste Tes
Role That Typicall Performs Test
Designer/Implementer
Developer or Test Engineer
Test Engineer/ Certification

Figure 13. Application by Roles

3.3 PROCESS DETAILS

An engineer develops a model for a component's requirements and interfaces and generates tests from it. The test cases are then transformed by the test driver generator into test scripts (aka test drivers) for automated test execution. Test engineers work in parallel with requirement and design engineers to refine the requirements and model them to support automated test design and test execution. The following list outlines the process, as depicted in Figure 14:

1. Working from whatever requirements artifacts are available, testers or modelers create models using the TTM tool based on the SCR method. Tables in the model represent each output, specifying the relationship between input values and resulting output values. The tools check the models for inconsistencies. The modeler interacts with the requirements engineers to validate that the model is a complete and correct interpretation of the requirements.
2. The tester maps the variables (inputs and outputs) of the model to the interfaces of the system in object mappings. The nature of these interfaces depends on the level of testing performed. See Section 3.4.3 for additional details.
3. The T-VEC tool generates test vectors for testing each (alternative) path in the model. These test vectors include test inputs and expected test outputs, as well as model-to-test traceability.
4. T-VEC generates the test drivers using the object mappings and schema. A schema is created once for each test environment. The schema defines the algorithmic pattern to carry out the execution of the test cases. The test driver executes in the target, host, or simulation environment. The test drivers typically are designed as an automated test script that sets up the test inputs enumerated in each test vector, invokes the element under test, and captures the results.
5. T-VEC analyzes the test results by comparing the actual test results to the expected results. T- then highlighting any discrepancies in a summary report.

Consortium members have applied this conceptual process in modeling many different types of applications. This is the basic approach that underlies all the case studies discussed in this report.

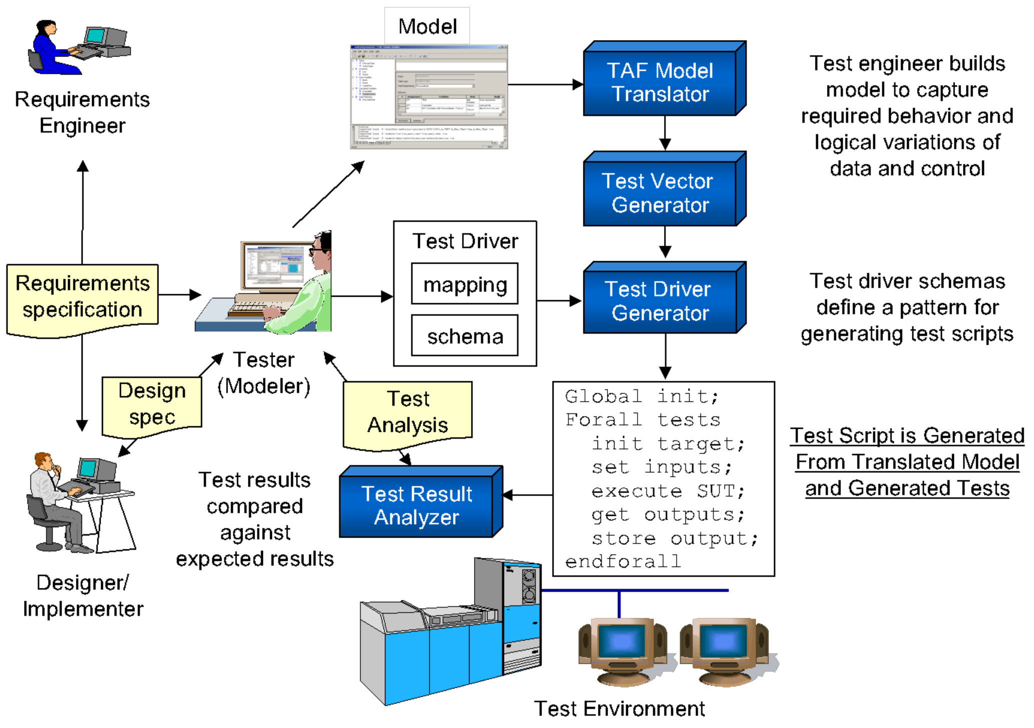


Figure 14. Model-Based Test Automation

3.4 GENERAL GUIDELINES FOR DEVELOPING A MODEL

This section provides some general guidelines for developing requirement models. Figure 15 shows a conceptual recursive process that is applied by the team of requirement, design, and modeling engineers to a component at any level of a system. This process does not imply a waterfall modeling approach. It is based on a process whereby implementation-derived requirements from higher-level design decisions manifest themselves in an operational context for starting the modeling process at a lower level (i.e., related to implementation-derived requirements). The recursive process steps are driven by the modeling tasks. Once a version of the model is completed, even for a thread of the component functionality, the testing can be performed at that level. This approach addresses implementation-derived requirements allocated to components at various layers of the system architecture and reduces the complexity of verification.

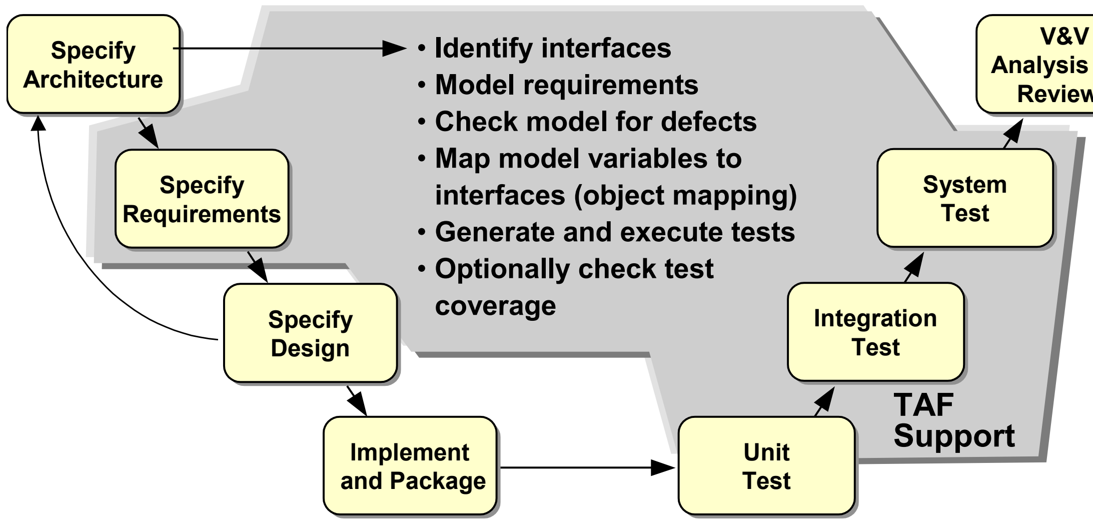


Figure 15. Recursive V-Model

The Help menu of the TTM tool provides details to perform the tasks in Sections 3.4.1 through 3.4.3.

3.4.1 Define the Interfaces for the Model

- Identify the interface boundaries of the component; the architecture at any level of the system is the context for the component under test.
- Create new types and constants whenever they are needed in the course of the model development.
- Identify the input (monitored) and output (controlled) variables:
  - Identify modes and terms (See Appendix 0 for details).
- Define the variables:
  - Define types for numeric variables so that the legal range of values can be specified.
  - Define types for enumerated variables.
  - Define Boolean variables (e.g., a flag) directly.

3.4.2 Define Behavioral Model Elements

Use a goal-oriented approach, and work backward by identifying each output (controlled variable or term) of the component:

- Create a table that assigns the value for each different computed value of the output.
- Use a condition table to describe relationships between outputs if the relationships are continuous over time (i.e., invariant over time).
  - Work backward, finding all of the conditions that must be TRUE for the function related to the output to be relevant. See Conditions in Section 0.

- Use an event table (with mode or modeless) to describe relationships between an output (or term) if the relationships are defined at a specific point in time.
  - Define the events and optional guard conditions that trigger the event. See Events in Section 0.
- Use a mode transition table (similar to a state machine) to describe relationships between an object if the relationship for a mode is defined for a specific interval of time (set of related system states):
  - Identify the set of modes; define the event associated with each source-to-destination transition.

If there are common conditions that are related to constraints (i.e., conditions or events) of functions of two or more outputs (or terms), then define a term table that can be referenced in all relevant tables.

3.4.3 MAP THE MODEL TO IMPLEMENTATION INTERFACES

The second aspect in the process to support test automation deals with mapping the model variables to the corresponding implementation interfaces of the component to be tested. As shown in Figure 16, the verification model includes the following:

- TTM models that are defined through analysis and interpretation of the requirements; this process is iterative, where tool analysis continues to help a modeler formalize details and remove inconsistencies or contradictions within the model requirements.
- Object mappings that relate input and output variables to the interfaces of the designed or implemented system. At the system level, the interfaces may include GUI widgets, database APIs, or hardware interfaces. At the lowest level, they can include class interfaces or library APIs. The tester uses these object mappings with a test driver pattern (aka schema) to support automated test script generation. The tester works with the designers to ensure the validity of the interface mappings from model to implementation.
- The test driver schema defines the test-environment-specific details for generating test scripts, procedures, or test harness elements for initializing the target, injecting inputs, executing the component under test, and extracting the output. NOTE: A test driver schema needs to be defined only once for a test environment.

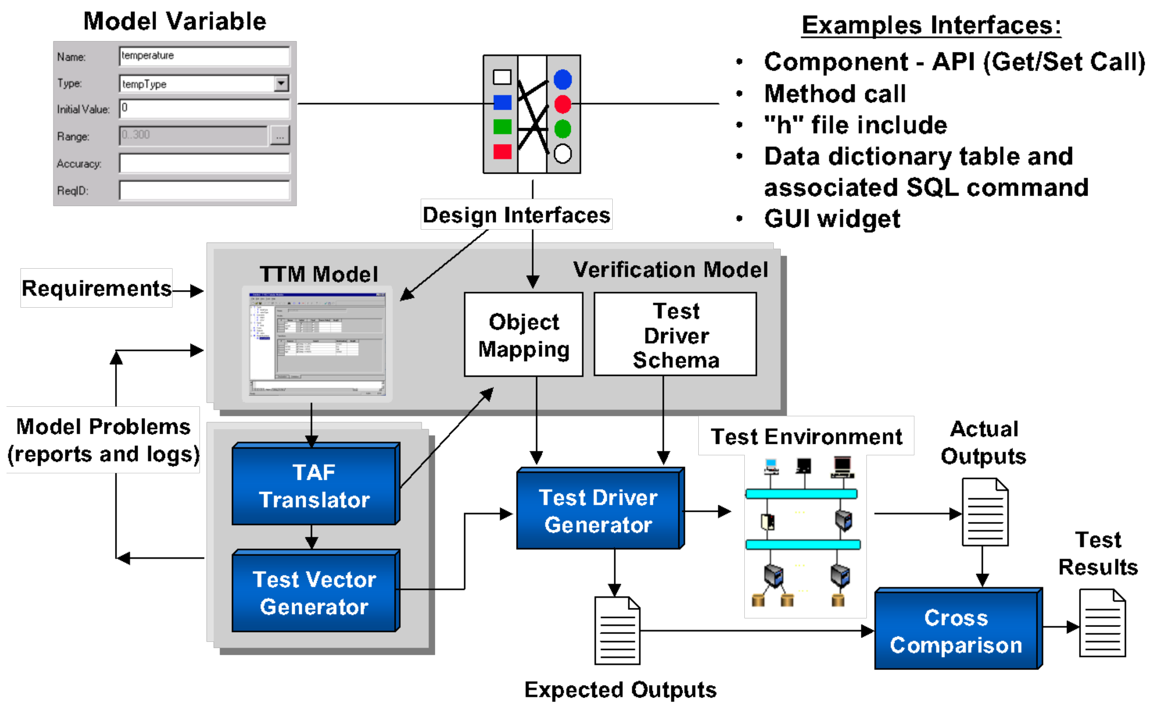
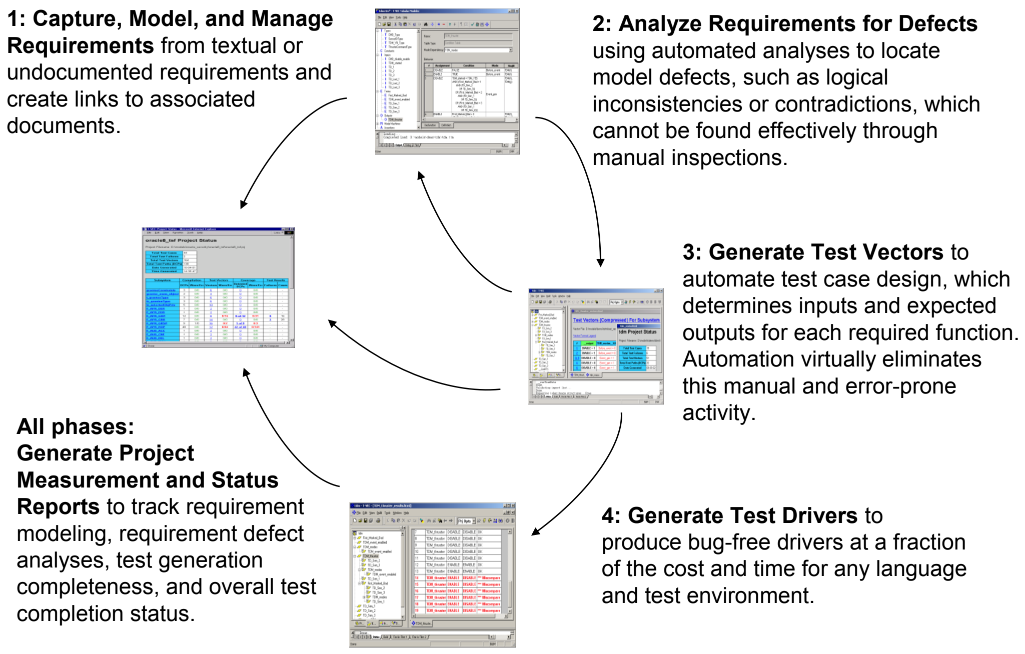


Figure 16. Verification Model Elements and Tool Relationships

3.5 TOOL PROCESS SUMMARY

As reflected in Figure 16 and Figure 17, test vectors are generated from the modeled information. During the test generation process, model checking is performed automatically, and defects such as logical contradictions are identified. The modeler must correct the problems in order to get a complete set of tests for the model. Once the model is correct and the object mappings are completed, test drivers are generated that can execute against the implementation. The expected outputs are compared against the actual outputs, and a test results report is generated along with project measurement information.



1: Capture, Model, and Manage Requirements from textual or undocumented requirements and create links to associated documents.

2: Analyze Requirements for Defects using automated analyses to locate model defects, such as logical inconsistencies or contradictions, which cannot be found effectively through manual inspections.

3: Generate Test Vectors to automate test case design, which determines inputs and expected outputs for each required function. Automation virtually eliminates this manual and error-prone activity.

4: Generate Test Drivers to produce bug-free drivers at a fraction of the cost and time for any language and test environment.

All phases: Generate Project Measurement and Status Reports to track requirement modeling, requirement defect analyses, test generation completeness, and overall test completion status.

Figure 17. Tool-Supported Processes

3.6 SUMMARY

A constructive and layered approach to development and verification helps reduce the cost of systematic verification. Layered verification allows the number of test cases to have a linear relationship with the number of paths in the set of system components. Components can be hierarchically related to support integration testing of a high-level subsystem without requiring tests for each referenced lower-level component. This approach precludes the combinatorial explosion associated with attempting to create tests from the combination of constraints associated with each path through a hierarchy of components. This process also helps guide the design of components that are easier to test because it promotes the development of interfaces at each level of the system, which provides a more testable design. The TAF model-based testing approach has been applied by Consortium members to many different levels of systems. The case studies in Sections 4 through 14 cover the spectrum of applications in which TAF has been applied using the general approach covered in this section.

4. MARS POLAR LANDER

4.1 PROBLEM

The MPL project started February 1994. The MPL was lost on December 3, 1999, after 11 months in space, having traveled at least 35 million miles, with a cost of approximately $165 million; it was only 40 meters from landing. Because of a bug, the Touchdown Monitor (TDM) software falsely indicated landing, causing a premature engine shutdown and subsequent MPL crash. The verification activities were comprehensive and performed by dedicated, experienced engineers. According to Bob Knickerbocker (Director of Software at Lockheed Martin Space Systems Company Astronautics Operations), Lockheed Martin had serendipitously found the bug a couple months after the crash.

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. The TDM should have ignored this spurious indication, but because of a design flaw, the state of the leg sensor signals was "latched" in the TDM computer memory, thus causing the premature engine shutdown.

Lockheed Martin was responsible for the development and verification of the MPL spacecraft and on-board software. The Lockheed Martin team was knowledgeable about the TAF capabilities through use by other Lockheed Martin companies [Safford 2000]. They wanted to know whether the use of TAF would have found the fault. Lockheed Martin sent the requirements and the code to the SSCI 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.

4.2 APPROACH

The TAF team deliberately did not look at the code before creating the tests. Rather, they created a model, derived form the English-language requirements, using the SCR tool developed by the Naval Research Laboratory. Figure 18 shows the conceptual process flow that relates the artifacts to the tools. The TDM specification is modeled using the SCR tool from the single page of textual MPL touchdown landing requirements shown in Figure 20, without knowledge of the code failure, and with no other information and support from Lockheed Martin. TAF 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.

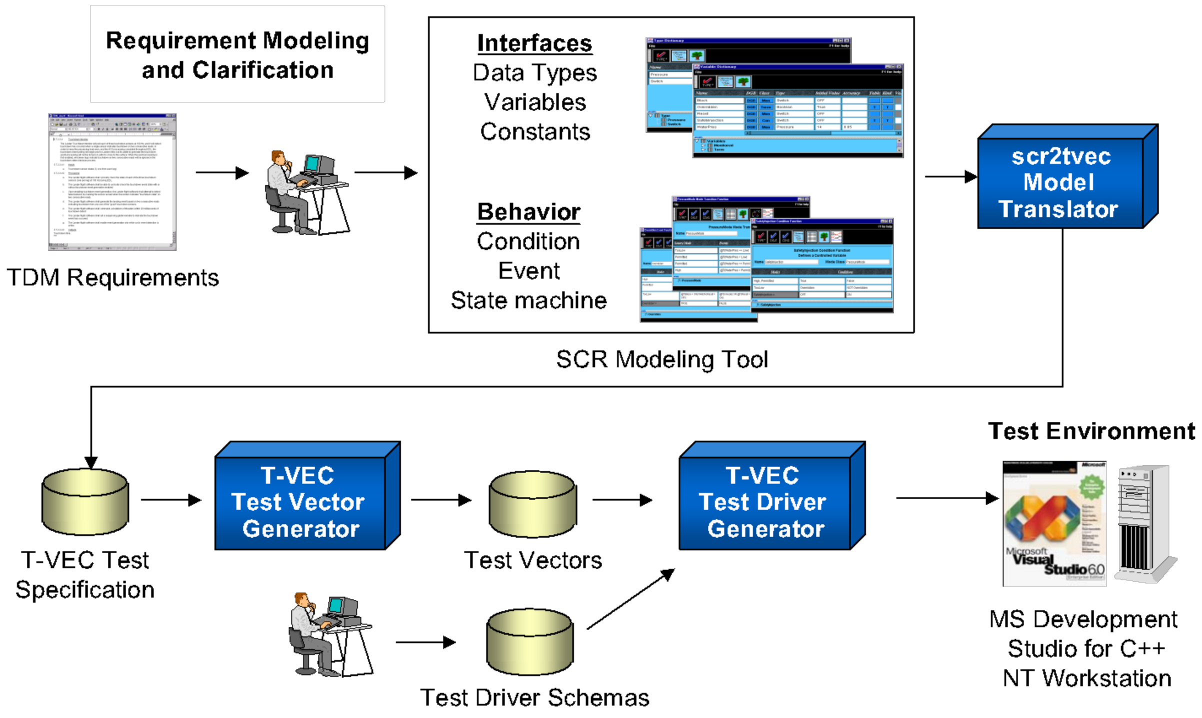


Figure 18. Process Flow and Artifacts

4.3 IMPLEMENTATION

4.3.1 INTENDED BEHAVIOR

The software requirements are paraphrased as follows (see actual requirements in Figure 20), but Lockheed Martin provided some additional insights almost 2 years after discovery of the fault. Figure 19 shows details relating to the failure scenario and the TDM component interfaces.

The TDM is a software component of the MPL system that monitors the state of three landing legs during two stages of the descent. As shown in Figure 19, the interfaces to the TDM module include a real-time, multitasking executive and leg sensors. The executive calls the TDM at a rate of 100 times per second to read the sensor leg data for each of the three legs.

During the first stage, starting approximately 5 kilometers above the Mars surface, the TDM software monitors the three touchdown legs. One sensor for each leg is used to determine whether the leg touched down. When the legs lock into the deployed position, there was a known possibility that the sensor might indicate a touchdown signal. The TDM software was supposed to handle this potential event by marking a leg that generates a spurious signal on two consecutive sensor-reads as having a “bad” sensor. During the second stage, 40 meters above the Mars surface, the TDM software was supposed to monitor the remaining “good” sensors. When a sensor had two consecutive reads indicating touchdown, the TDM software was supposed to command the descent engine to shutdown. There is no absolute way to confirm what happened to the MPL, but the following is believed to be the failure scenario.

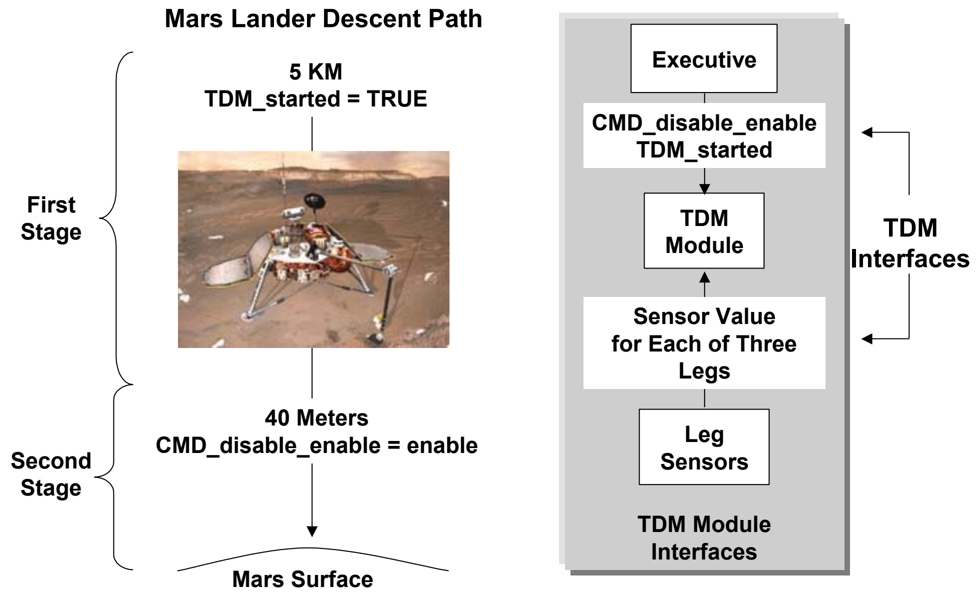


Figure 19. Mars Polar Lander Details

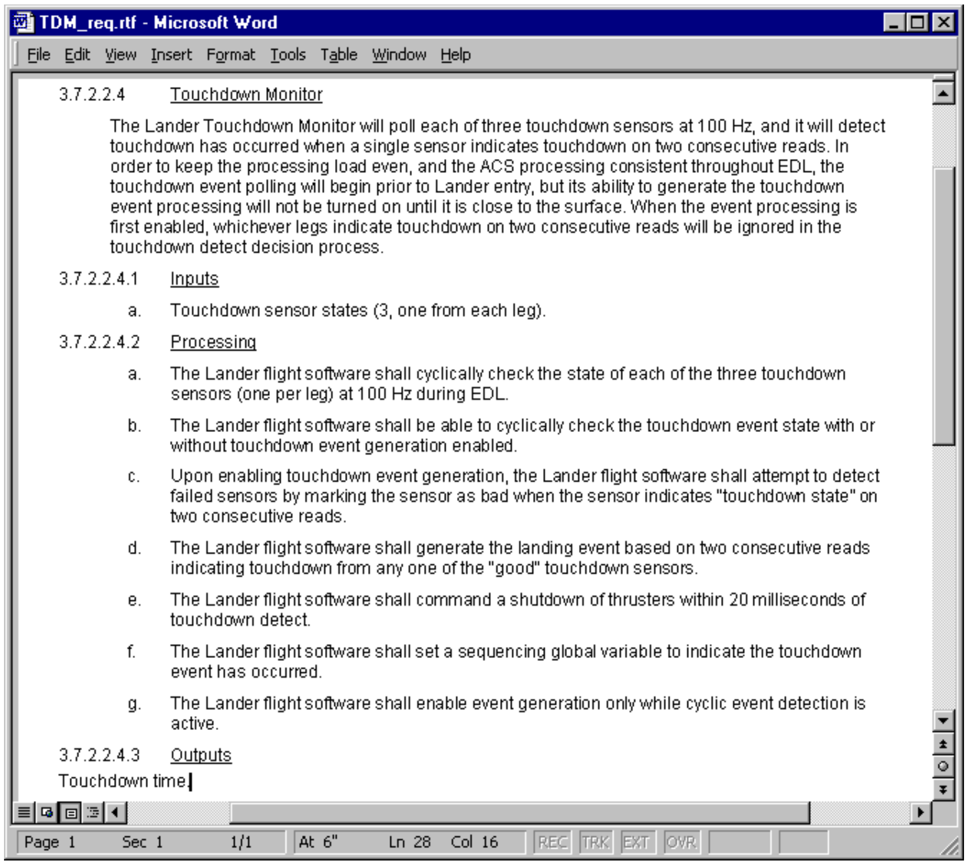
The MPL was in the first stage of descent (5 kilometers). The engine was on.

1. The landing legs were deployed and locked into position.
2. During a clock tick, one of the legs (e.g., leg 1) showed an incorrect touchdown indication. That touchdown indication was stored in a program variable. Call it sensor[1].
3. On the next clock tick, the value of sensor[1] was copied into last_sensor[1]. That variable tells whether a touchdown indication was seen in the previous clock tick.
4. The same leg still showed a touchdown indication. That indication was stored in sensor[1]. Because both sensor[1] and last_sensor[1] were set, further sampling from leg 1 was turned off. However, *the variables retained their values*.
5. When the MPL entered the second stage of descent (40 meters), the processing of leg touchdown indications changed, and the MPL should have turned off the engine when the "sensor" and "last_sensor" variables for any leg (provided the leg had not been marked bad) both indicated a touchdown event.
6. The failure occurred because sensor[1] and last_sensor[1] indicated a touchdown, so the engine was erroneously turned off approximately 40 meters above the surface instead of on touchdown.

There are many ways that the requirement could have been designed and implemented, but the essence of the design flaw is that the program variables retained the state of the “bad” sensor information.

4.3.2 TDM Requirements and Model

Lockheed Martin supplied the textual requirements shown in Figure 20 to the TAF team. Developing SCR models requires identifying the system input and output variables and defining the relationships between them. Typically, this process is iterative. It involves defining the variables, data types associated with the variables, and the tables that define relationships between the variables. 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.


TDM_req.rtf - Microsoft Word
File Edit View Insert Format Tools Table Window Help
Page 1 Sec 1 1/1 At 6" Ln 28 Col 16 REC TRK EXT OVR

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 20. TDM Requirements (From Lockheed Martin)

The 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.

Although the requirements document indicates that the output is "Touchdown time," the key output associated with the code interface 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.

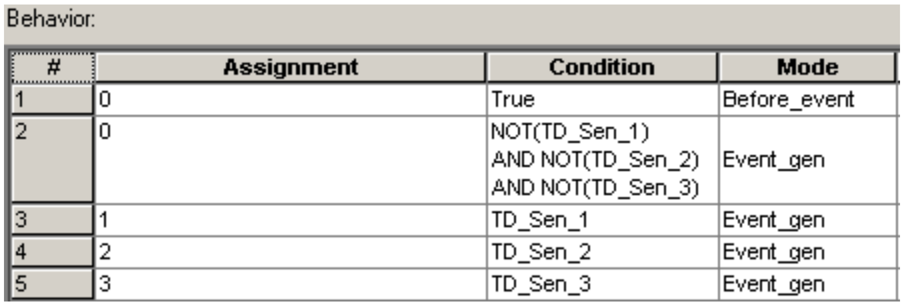
4.3.3 EXAMPLE MODELS

Once the system's interfaces are defined, its behavior is modeled in SCR using condition tables and one mode table. These tables define the value of a variable in terms of input, terms (intermediate), and mode (state) variables. A condition table defines the output value for TDM_thruster. It depends on five condition tables and one mode table.

A mode table, TDM_Modes defines two modes, relating to descent stages. The mode Before_event is the mode associated with the descent between 5 kilometers to 40 meters. The mode Event_gen, related to the requirement referred to as "touchdown event generation," is the stage of descent that starts approximately 40 meters above the surface of Mars.

Three condition tables named TD_Sen_1, TD_Sen_2, and TD_Sen_3 define the conditions associated with the sensor signal for each landing leg. The condition table First_Marked_Bad 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. The condition table TDM_thruster relies on the output of the other tables to specify the behavior for the value of the output TDM_thruster.

Two examples tables provide details of the model. The term First_Marked_Bad, shown in Figure 21, is modeled as an Integer that returns a value between 0 and 3. The table First_Marked_Bad is also associated with the mode table TDM_Modes. The first column of the table contains the value 0 to 3. The second column describes the conditions. The third column defines the mode, 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 0, 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 1, 2 or 3 depending on the condition associated with the term for the sensors TD_Sen_1, TD_Sen_2, or TD_Sen_3; otherwise, it takes on the value 0.


Behavior:

#	Assignment	Condition	Mode
1	0	True	Before_event
2	0	NOT(TD_Sen_1) AND NOT(TD_Sen_2) AND NOT(TD_Sen_3)	Event_gen
3	1	TD_Sen_1	Event_gen
4	2	TD_Sen_2	Event_gen
5	3	TD_Sen_3	Event_gen

Figure 21. Behavioral Specification for First_Marked_Bad

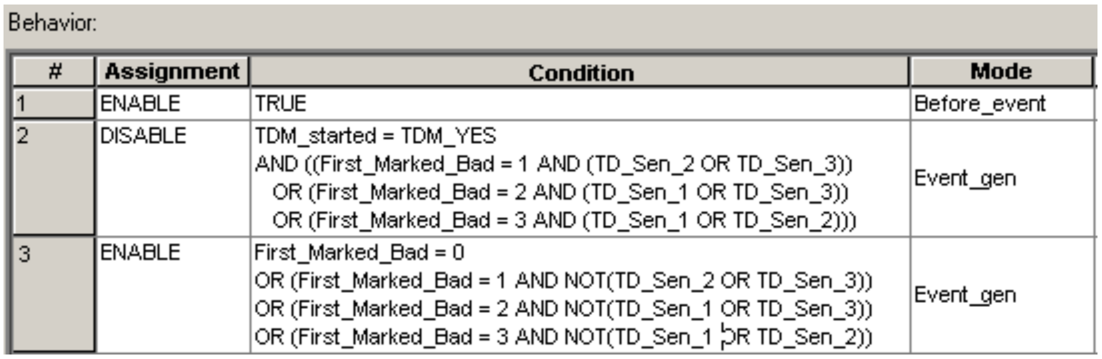
Figure 22 shows the condition table for TDM_thruster. Like First_Marked_Bad, TDM_thruster also is associated with the mode table TDM_Modes. When the mode is Before_event, the thruster always must be ENABLE. After the Event_gen, the thruster takes on the value DISABLE when TDM_started is equal to TDM_YES, 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 0 – 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.

Row 3 illustrates the specification of the situation that identified the bug. It represents the case where one of the legs was marked bad, but no other sensor reads occurred. This approach reflects the recommended practice of specifying the positive cases as well as the negative cases.


Behavior:

#	Assignment	Condition	Mode
1	ENABLE	TRUE	Before_event
2	DISABLE	TDM_started = TDM_YES AND ((First_Marked_Bad = 1 AND (TD_Sen_2 OR TD_Sen_3)) OR (First_Marked_Bad = 2 AND (TD_Sen_1 OR TD_Sen_3)) OR (First_Marked_Bad = 3 AND (TD_Sen_1 OR TD_Sen_2)))	Event_gen
3	ENABLE	First_Marked_Bad = 0 OR (First_Marked_Bad = 1 AND NOT(TD_Sen_2 OR TD_Sen_3)) OR (First_Marked_Bad = 2 AND NOT(TD_Sen_1 OR TD_Sen_3)) OR (First_Marked_Bad = 3 AND NOT(TD_Sen_1 OR TD_Sen_2))	Event_gen

Figure 22. Behavior Specification for TDM_thruster

4.3.4 TEST VECTOR GENERATION

The T-VEC tool converts a model into test vectors that exercise the conditions in the model, special values of the variables (like boundary values for floating point variables), and special combinations of values. Table 2 shows the 19 vectors derived from the model. TDM_thruster is the output variable; it defines the expected output for the engine. CMD_disable_enable defines whether the MPL can turn off the thruster upon appropriate signal from the legs; it is disabled above 40 meters and enabled at or below 40 meters. TD_1 and TD_Last_1 show whether the leg registered touchdown on the current and previous clock ticks. First_Marked_Bad shows which leg was marked bad, if any; if none were marked bad, it contains 0. The tool also combines redundant test vectors, like 3 and 5, to reduce the size of the test set.

Tests 17 through 22 are test vectors that expose the failure. In test 22, engine shutdown is enabled. Legs 1 and 2 are not signaling touchdown. TD_3 and TD_Last_3 show that the leg has signaled touchdown in two clock ticks. However, First_Marked_Bad shows that the leg sensor is considered bad. So the expected result is to leave the engine enabled. As is known from the failure scenario, the software will instead shut the engine down.

4.3.5 TEST DRIVER GENERATION AND EXECUTION

Executing the test requires creating a test driver. The test driver generator combines test vectors and object mapping, which map the model variable to the implementation variables or interfaces to produce test drivers. The test vector shows eight inputs, but only five of them are true inputs to the software. TD_Last_1, TD_Last_2, and TD_Last_3 are not inputs. Rather, they are ways of noting that the software, to meet its requirements, must store the previous values of the inputs TD_1, TD_2, and TD_3. So, each test vector is executed in two steps. In the first, the values of TD_Last_1, TD_Last_2, and TD_Last_3 are provided. In the second, the values of TD_1, TD_2, and TD_3 (as well as CMD_disable_enable) are provided to represent the next clock tick, and the actual output is checked against the expected output.

The test driver also uses a test driver schema, which encodes an algorithmic pattern for test execution for the specific test environment. 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 simple language that accesses information about the test vectors. 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. The generated test drivers were executed programmatically on a Windows NT platform. The test driver schema simulates the way the multitasking executive called the TDM entry point. This approach for calling the subsystem under test is commonly used with other TAF schemas to propagate state data. The test driver made two calls to the main entry point. The execution of the test driver resulted in a failure that emulated the situation where state data would propagate and latch into a particular state. The tests, shown in Table 2, which uncovered the failure scenario, are associated with the modeled requirement First_Marked_Bad, defined in Figure 21.

Table 2. Test Vectors for TDM_thruster

Test ID	Expected Output	Inputs							
	TDM_thruster	First_Marked_Bad	CMD_disable_enable	TD_1	TD_Last_1	TD_2	TD_Last_2	TD_3	TD_Last_3
1	ENABLE	3	DISABLE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE
2	ENABLE	0	DISABLE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE
3.5	DISABLE	1	ENABLE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE
4	DISABLE	1	ENABLE	TRUE	TRUE	TRUE	TRUE	FALSE	FALSE
6	DISABLE	1	ENABLE	TRUE	TRUE	FALSE	FALSE	TRUE	TRUE
7.9	DISABLE	2	ENABLE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE
8	DISABLE	2	ENABLE	TRUE	TRUE	TRUE	TRUE	FALSE	FALSE
10	DISABLE	2	ENABLE	FALSE	FALSE	TRUE	TRUE	TRUE	TRUE
11.13	DISABLE	3	ENABLE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE
12	DISABLE	3	ENABLE	TRUE	TRUE	FALSE	FALSE	TRUE	TRUE
14	DISABLE	3	ENABLE	FALSE	FALSE	TRUE	TRUE	TRUE	TRUE
15	ENABLE	0	ENABLE	FALSE	TRUE	FALSE	TRUE	FALSE	TRUE
16	ENABLE	0	ENABLE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE
17	ENABLE	1	ENALBE	TRUE	TRUE	FALSE	TRUE	FALSE	TRUE
18	ENABLE	1	ENABLE	TRUE	TRUE	FALSE	FALSE	FALSE	FALSE
19	ENABLE	2	ENABLE	FALSE	TRUE	TRUE	TRUE	FALSE	TRUE
20	ENABLE	2	ENABLE	FALSE	FALSE	TRUE	TRUE	FALSE	FALSE
21	ENABLE	3	ENABLE	FALSE	TRUE	FALSE	TRUE	TRUE	TRUE
22	ENABLE	3	ENABLE	FALSE	FALSE	FALSE	FALSE	TRUE	TRUE

4.4 KEY GUIDELINES

4.4.1 USE GOAL-ORIENTED MODELING, WORKING BACKWARD FROM OUTPUTS

This case study reflects the recommended process for using the TAF as discussed in Section 3.3. The modeling process is goal-oriented, working backward from the output (TDM_Thruster). The process starts from TDM_Thruster and describes the conditions associated with each value that it can take on. Common conditions such as First_Marked_Bad are defined in a separate table and referenced several times.

4.4.2 CHARACTERIZE CRITICAL BEHAVIOR IN MODELS

This case study illustrates an important point: models do not have to characterize every possible combination of behavior to be effective in identifying faults. For example, the model does not have a situation where two legs are marked bad prior to the event generation event. However, it is common for the number of tests generated from a model to significantly exceed the number of test cases created manually.

4.4.3 MODEL BOTH POSITIVE AND NEGATIVE CASES

It is necessary to specify the required functional behavior, and most manual test cases cover the common or positive case. However, sometimes testers forget to specify the negative case such as that illustrated in Row 3 of Figure 22. In this case, a sensor leg is marked bad, but where there were no sensor indications on the other legs that were not bad. This part of the model produced the tests that identified a bug in the implementation of the TDM. These types of cases often can identify unhandled situations in the code implementation.

4.4.4 Ensure That Test Driver Propagates State Data

When testing components with state data, the test driver must make two or more calls to the component under test to cause the input data to propagate through the state or historical data of the implemented system so that it is observable as an output. The test driver schema is the place where this type of pattern is implemented once but can be applied for all tests related to the test environment. The following algorithm illustrates this point given the assumption that the state information is only one level deep like the TDM (i.e., the monitor observes two reads on any one sensor: the current value and the previous value).

```
1  Loop through all test cases
2   Initialize the test environment
3   Set the output to a value other than expected
4   Set the inputs
5   Call the system under test (to set the historical state data)
6   Set the inputs
7   Call the system under test
8   Get the output
9  End loop
```

Line 5 is the critical addition to this pattern. It causes the inputs to be propagated into the state or historical data; the second time the subsystem is executed, the state data combined with the input data are used to compute the output. In the case of the TDM, this resulted in a failure. The state history in the TDM application is only one-deep because the software monitored two consecutive reads of the leg sensor data. If the state information has more states (e.g., current data, previous data, and second previous data), then the propagation would require additional calls or processing like Line 5 in the algorithm.

4.5 RESULTS

In fewer than 12 hours, the TAF model-based testing approach identified the code bug that was the probable cause of failure, without the support of Lockheed Martin. Lockheed Martin's Bob Knickerbocker stated that T-VEC's systematic approach, supported by the tools, provides a standardized test approach and a more thorough test capability than the manual approach. The test driver generation mechanism provides the flexibility to simulate the real-time environment of the TDM code module. These results suggest that TAF provides more standardized and thorough testing for verification of critical software and system functionality. TAF provides the capabilities to identify critical software and system defects to significantly minimize the risk where one bug can be catastrophic as in the case of the MPL failure.

The results of the application suggest that the TAF approach has the potential to provide a systematic and cost-effective approach for verification. Lockheed Martin believes the tool provides a standardized test approach and a more thorough test capability than the manual approach. Lockheed Martin has used TAF on other projects [Boden 2004].

This page intentionally left blank.

5. PRINTER FEATURE PROCESSING

5.1 PROBLEM

The application for this case study is based on testing the software that implements the XPIF. The XPIF is a mapping of the common print semantics specification into eXtensible Markup Language (XML). The specific project focuses on a reduced set of XPIF that complies with the syntactical structures defined in the IPP[5]. XPIF is processed by component applications that control the way documents should be printed and processed. There are a large number of combinations to test, but there are different variations that must be supported by different types of printers. In addition, the specification features continue to expand, requiring significant maintenance to the test and significant regression testing. This is expensive to do manually and often creates only subsets of the tests.

5.2 APPROACH

Figure 23 illustrates the conceptual environment and interfaces of the XPIF processing. Like many SSCI members, this member has a printer engine test execution tool that can automate the test execution. It also provides a pseudo-target environment to load and execute different versions of the software because the actual printer hardware and software were being developed concurrent with this application of TAF.

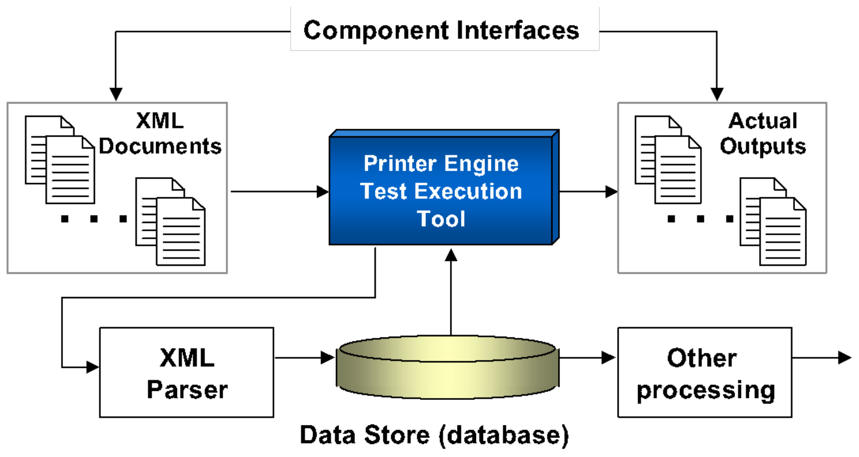


Figure 23. XPIF Test Execution Environment

The objective is to produce different combinations of XPIF printer specifications that test the various combinations of operation and job-template attributes. An XPIF document is input to the

[5] http://ietf.org/rfc/rfc2566.txt

printer engine test execution tool in XML format. The component under test parses the XPIF document, and attributes parsed from the XPIF are stored in a database (referred to as the Data Store). Other functions produce the actual documents read from the Data Store to produce the document. The test execution tool can access one or more XPIF files, submit them for processing, suspend the processing temporarily while it retrieves the desired attributes from the Data Store, and then write those attribute names and values to an actual output file.

To automate testing fully, it is critical to understand the interfaces of the component as well as the test environment prior to modeling because these interfaces are the model inputs and outputs. In addition, the input and output representations must be known in order to define the object mappings because it is critical that these inputs are programmatically “settable” and the outputs are programmatically “gettable” to support full test automation without manual intervention.

5.3 IMPLEMENTATION

Figure 24 provides a perspective of the process used to model and generate test scripts for testing the XPIF parsing function. The primary input to the modeling process is the XPIF specification document and an XML Document Type Declaration (DTD) [http://www.w3.org/TR/REC-xml/] definition that provides details for the various attributes that should be modeled, and their representation produces an XML file.

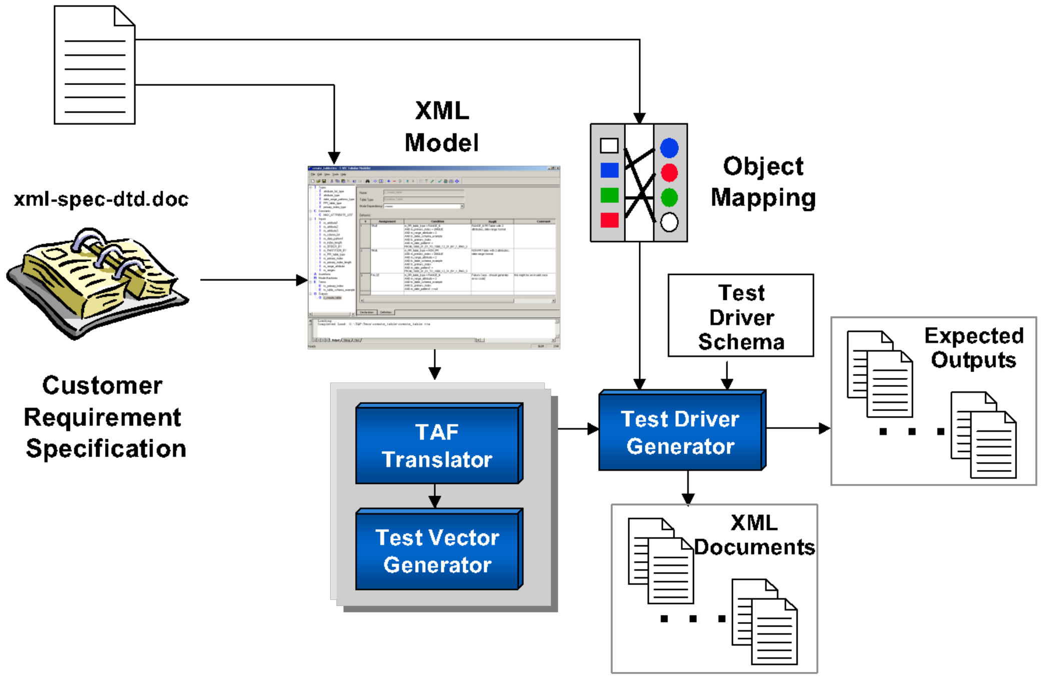


Figure 24. XPIF Modeling Scenario

An XPIF document has several components:

- Xml version
- Document Type

- Xpif version and section that contains:
  - xpif-operation-attributes section with many possible attributes
  - job-template-attributes section with many possible attributes

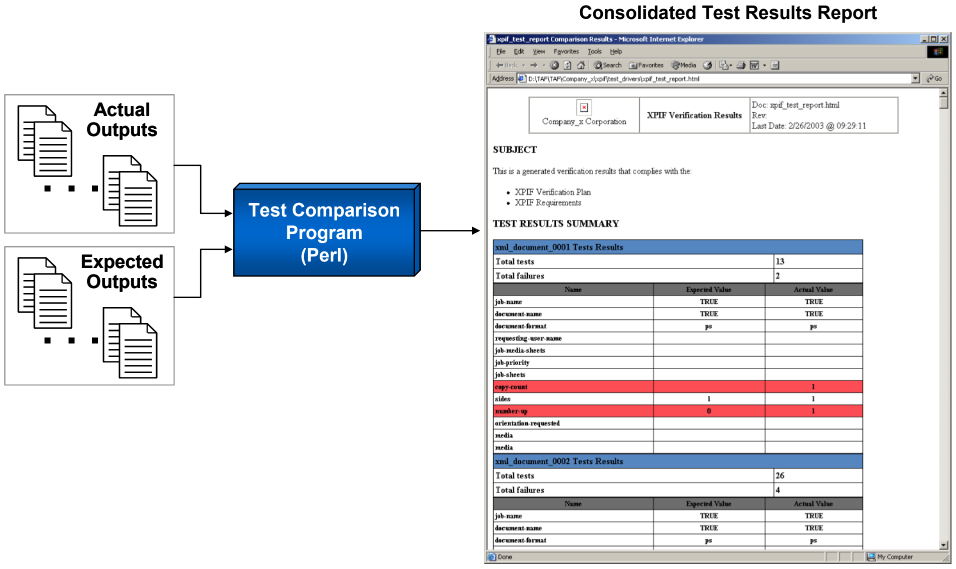
Some example attributes specified by XPIF include job-priority, multiple-document-handling, copies, page-ranges, orientation-requested, print-quality. The following is an example of the XML specifying some XPIF attributes:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE xpif SYSTEM "xpif-v02012.dtd">
<xpif cpss-version="2.01" version="1.0" xml:lang="en">
    <xpif-operation-attributes>
        <job-name syntax="name" xml:lang="en"
         xml:space="preserve">tiger.ps</job-name>
        <requesting-user-name syntax="name"
         xml:space="preserve">ydufresn</requesting-user-name>
    </xpif-operation-attributes>
    <job-template-attributes>
        <copies syntax="integer">2</copies>
        <finishings syntax="1setOf">
            <value syntax="enum">3</value>
        </finishings>
        <job-recipient-name syntax="name"
         xml:space="preserve">cmiyachi</job-recipient-name>
        <media-col syntax="collection">
            <media-type syntax="keyword">stationery</media-type>
            <media-hole-count syntax="integer">0</media-hole-count>
            <media-color syntax="keyword">white</media-color>
            <media-size syntax="collection">
                <x-dimension syntax="integer">21600</x-dimension>
                <y-dimension syntax="integer">27900</y-dimension>
            </media-size>
        </media-col>
        <sheet-collate syntax="keyword">collated</sheet-collate>
        <sides syntax="keyword">one-sided</sides>
    </job-template-attributes>
</xpif>
```

The processed XPIF attributes have related attributes in the Data Store. The expected outputs are compared against the attribute values in the Data Store to ensure that each combination is processed properly. This approach provides a useful environment and more efficient process for testing because it is manually intensive to visually inspect each produced document for each test case to ensure that the proper processing is performed.

The TTM model for the XPIF function was translated, and test vectors were generated. The test vectors, combined with the object mappings and test driver schema, were input to the test driver generator to produce a set of XML files covering the various combinations of XPIF functions, one for each test case. The test driver produced expected outputs associated with the processed attributes stored in the Data Store for each corresponding XPIF file.

The test comparison and test results report generation checks that each attribute value in the actual test output file has the proper value as specified by the expected output file. In this case, each test vector was required to map to one XPIF file. There were many test vectors, resulting in many XPIF files. There was one actual output file and one expected output for each XPIF file. A Test Comparison program was tailored from a template in the TAF toolkit, which is written in Perl. It compared the results of each actual output file with those of the expected output file and generated a HyperText Markup Language (HTML) report where each output attribute value that did not compare with the expected output attribute value was highlighted in red as shown in Figure 25. The modeling and testing were performed simultaneously to the XPIF parser development. Test cases uncovered several XPIF attributes that were not completely implemented, as well as other defects in the program.


Consolidated Test Results Report
Actual Outputs
Expected Outputs
Test Comparison Program (Perl)
xpif_test_report Comparison Results - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites Media
Address D:\TAF\TAF\Company_x\xpif\test_drivers\xpif_test_report.html
Go
Done
My Computer

Company_x Corporation	XPIF Verification Results	Doc: xpif_test_report.html Rev: Last Date: 2/26/2003 @ 09:29:11

SUBJECT

This is a generated verification results that complies with the:

- XPIF Verification Plan
- XPIF Requirements

TEST RESULTS SUMMARY

xml_document_0001 Tests Results		
Total tests		13
Total failures		2
Name	Expected Value	Actual Value
job-name	TRUE	TRUE
document-name	TRUE	TRUE
document-format	ps	ps
requesting-user-name		
job-media-sheets		
job-priority		
job-sheets		
copy-count		1
sides	1	1
number-up	0	1
orientation-requested		
media		
media		

xml_document_0002 Tests Results		
Total tests		26
Total failures		4
Name	Expected Value	Actual Value
job-name	TRUE	TRUE
document-name	TRUE	TRUE
document-format	ps	ps

Figure 25. Test Results Report Generation

5.4 KEY GUIDELINES

5.4.1 UNDERSTAND THE INTERFACES AND TEST ENVIRONMENT

Understand the interfaces and test environment before starting to model because the model inputs must be associated with the programmably settable inputs and the programmably gettable outputs within the test environment. In this case, the inputs were XML files with many attributes, one for each test case. The outputs were attributes that were parsed from the XML inputs and extracted by the test tool from the Data Store and saved in a file.

5.4.2 SELECT A FEATURE THAT WILL CHANGE

It is best to select a feature that will likely continue to change because this provides the most leverage when using and demonstrating TAF. Use the TAF team to help develop models and test driver support for actual product testing, or identify another application and related feature that also might be considered for a project deployment. After the pilot project, the test infrastructure should be stable and reusable for other projects.

The first follow-on project should consider other small projects or system threads where TAF could be used to support the interface-driven approach; this seems to be a critical way to help foster better component architectures that have support for testability.

5.4.3 Ask the TAF Team for Advice

The TAF capabilities have continued to evolve over the last decade, but sometimes the specific test environment may require special processing. Remember to ask the TAF team for help or advice because they are likely to have something in the toolkit that can help speed the deployment process. The following list provides two examples:

- The TAF/T-VEC tools provide a test results comparison and report generation capabilities, but for this application, each test case was represented by one file that contained many attributes that were compared. The TAF team has a toolkit with various utilities. The team was able to quickly tailor an existing cross-comparison and test report generator written in Perl for this application.

- Typically, a test tool usually has primitive networking capabilities. The TAF team needed to build a small file transfer mechanism for downloading the XML test files and uploading the actual output files. Again, the team used a small Perl program from the toolkit to automate the file transfer using File Transfer Protocol (FTP).

5.5 RESULTS

The key highlights include:

- Example application modeling, test vector generation, test driver generation, and fully automated test execution and results analysis were conducted in 1.5 days.

- A subset of the model specification included 20 input attributes, 40 requirements, with a total of 80 test vectors associated with 1,040 test cases, with 380 test failures.

- TAF-generated tests were more comprehensive and systematic than manually developed tests in producing tests cases because they uncovered errors not known by the developer.

- TAF was demonstrated to apply to XPIF, where models developed in a Windows environment can generate XML tests files processed through a member’s test tool that is integrated with an existing test execution tool.

- Members noted that if TAF approach is applied during development, it would likely lead to better development of error handling and error messaging, as well as improved API interfaces.

- The product types are applicable for applying TAF, and there is a strong desire to do a better job of architecting products based on reusable components. The best-practice approach for applying TAF, based on interface-driven, model-based testing, would help in developing better component interfaces.

- The continuous verification approach supported by the TAF helps identify requirement defects earlier to reduce rework. The rework associated with requirement defects is a costly problem with this and most SSCI members. Continuous verification helps shorten cycle time by performing much of the testing in parallel with development.

- TAF provides requirement-to-test traceability, and this was currently lacking with this member's testing process.

One of the final points noted by the member company team during the review of this pilot application was that the use of TAF would be valuable in fostering the design of better system interfaces if it were used in parallel with development like it was on this pilot. The management and project leads noted that the lack of specific requirements often gives the developer too much latitude, and they create too many features that are not needed with the application. These additional features often delay the release schedule and also contribute to reduced quality because of feature interaction problems.

6. SQL EXTENSION LANGUAGE PROCESSING

6.1 PROBLEM

This member company builds a parallel database management system (DBMS) that processes various languages and command sets that go beyond the standard SQL [http://www.itl.nist.gov/fipspubs/fip127-2.htm]. The TAF team worked with the member on several custom language processing examples. Testing these applications is time-consuming because it is manually intensive to produce test cases and test scripts to cover all the combinations of the various types of languages and command sets that are continually evolving. In addition, the effort and cost increase because these tests must be evolved or regression tested for each new release of the system. This case study covers an application that processes an extension to a standard SQL command called CREATE TYPE.

6.2 APPROACH

The TAF team developed a model from the language grammar for the CREATE TYPE function, which is one of the User Defined Types function to illustrate how TAF can systematically produce tests to cover each combination of the function and related options. The CREATE TYPE statement defines a user-defined structured type that includes zero or more attributes. Following is an example of a CREATE TYPE statement:

```
CREATE TYPE address_t AS
   (STREET     VARCHAR(30),
    NUMBER     CHAR(15),
    CITY       VARCHAR(30),
    STATE      VARCHAR(10))
    NOT FINAL
    MODE DB2SQL
       METHOD SAMEZIP (addr address_t)
       RETURNS INTEGER
       LANGUAGE SQL
       DETERMINISTIC
       CONTAINS SQL
       NO EXTERNAL ACTION

       METHOD DISTANCE (address_t)
       RETURNS FLOAT
       LANGUAGE C
       DETERMINISTIC
       PARAMETER STYLE DB2SQL
       NO SQL
       NO EXTERNAL ACTION;
```

The objective is to model the language syntax and generate CREATE TYPE commands that cover all parameter combinations. There are different notations, both textual and graphic, to specify the syntax of a command. The Backus-Naur Form (BNF) [http://en.wikipedia.org/wiki/Backus-Naur_form] is a common notation used to represent language syntax. Following is some of the language syntax for the "<structured type create>" language:

```
<structured type create> ::=
CREATE TYPE <structured type name>
[ <subtype clause> ]
[ AS <member list> ]
[ <instantiable clause> ]
NOT FINAL
[ { <encryption> | <compress> } ]
[ <structured type method specification list> ]

<structured type name> ::= [ <database name> <period> ] <identifier>

<subtype clause> ::= UNDER <supertype name>

<supertype name> ::= <structured type name>

<member list> ::=
<left paren>
<attribute definition> [ { <comma> <attribute definition> }... ]
<right paren>

. . . MORE
```

In BNF nonterminal symbols, such as <structured type create>, are further decomposed. Everything after the "::=" is part of their definition. The "|" symbol means "or."

6.3 IMPLEMENTATION

The model closely parallels the structure of the BNF grammar and covers each language option required for the test cases. There is a simple pattern for modeling language syntax defined in BNF or other related syntax language descriptions. See Section 11.4 for more details and specific guidelines. The test driver produces a sequence of SQL commands covering each of the different cases that must be processed by the target application. The SQL language statements are output to a test script language that is used to inject the SQL commands into the target test environment. The script language has special directives to initialize the different database environment for different types of processing scenarios as shown in Figure 26.

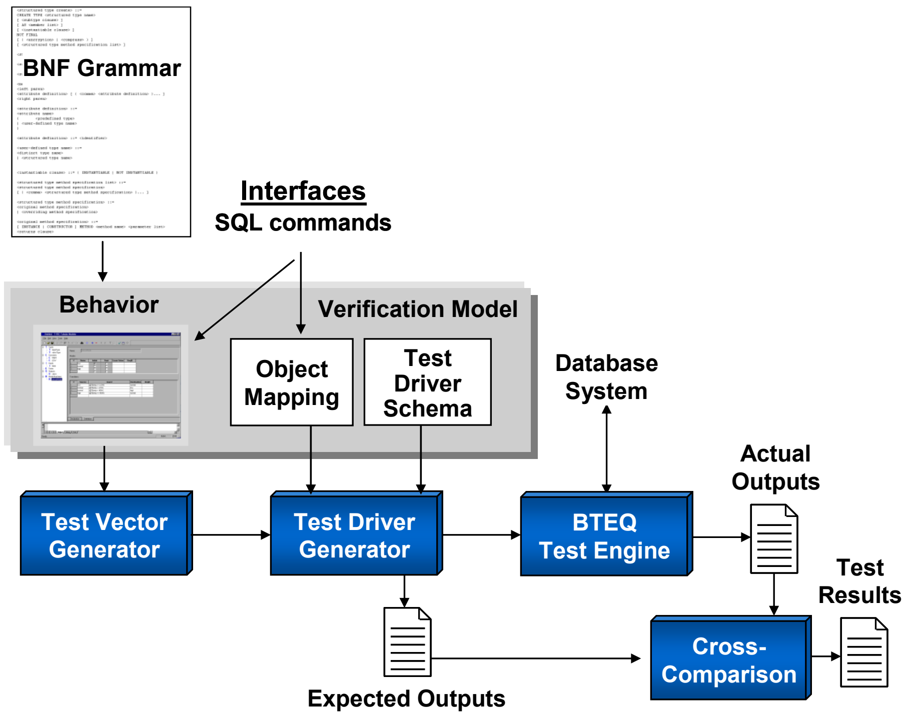


Figure 26. SQL Extension Test Environment

6.4 KEY GUIDELINES

There is a general pattern for modeling a language grammar using SCR condition tables within TTM. The generated tests provide a mechanism for testing command line functions with various parameters and options, language commands and data structures.

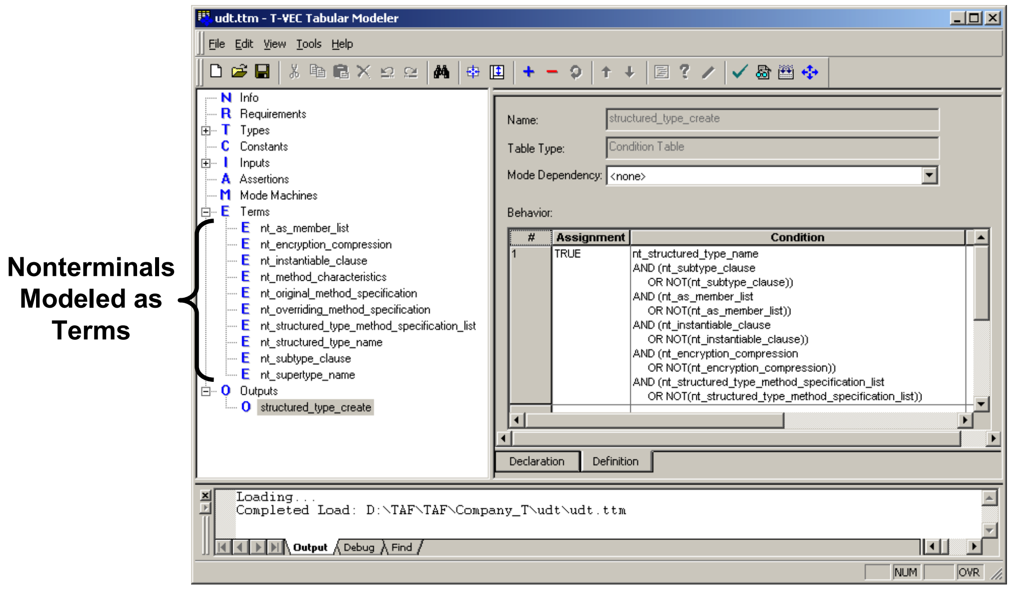
The syntax for the structured_type_create grammar has one fixed part of the function (structured_type_name) and six optional parts (e.g., subtype_clause, and as_member_list). The brackets identify parts that are optional in the following higher-level representation of the BNF syntax for structured_type_create.

```
<structured_type_create> ::= <structured_type_name>
    [ <subtype_clause> ]
    [ <as_member_list> ]
    [ <instantiable_clause> ]
    [ <instantiable_clause> ]
    [ <encryption_compression> ]
    [ <structured_type_method_specification_list> ]
```

Figure 27 shows the model representation for structured_type_create. It is modeled as a Boolean type because the outputs are simply the representation of the function command parts. Each nonterminal symbol of the grammar is modeled as a Term condition table, and an “ns_” is prefixed on the name to indicate that it is a nonterminal symbol. A non-terminal symbol is used when the language for that option has several elements. The approach used to specify that a symbol is optional is illustrated with the subtype_clause:

```
(nt_subtype_clause
            OR NOT(nt_subtype_clause))
```

This forces one vector to include the subtype clause when it is TRUE and when its value is FALSE that part of the clause is blank. This pattern is used for all of the optional components of the structured_type_create syntax.


Nonterminals Modeled as Terms
udt.ttm - T-VEC Tabular Modeler
File Edit View Tools Help
N Info
R Requirements
T Types
C Constants
I Inputs
A Assertions
M Mode Machines
E Terms
E nt_as_member_list
E nt_encryption_compression
E nt_instantiable_clause
E nt_method_characteristics
E nt_original_method_specification
E nt_overriding_method_specification
E nt_structured_type_method_specification_list
E nt_structured_type_name
E nt_subtype_clause
E nt_supertype_name
O Outputs
O structured_type_create
Declaration
Definition
Output
Debug
Find
NUM
OVR

Name: structured_type_create

Table Type: Condition Table

Mode Dependency: <none>

Behavior:

#	Assignment	Condition
1	TRUE	nt_structured_type_name AND (nt_subtype_clause OR NOT(nt_subtype_clause)) AND (nt_as_member_list OR NOT(nt_as_member_list)) AND (nt_instantiable_clause OR NOT(nt_instantiable_clause)) AND (nt_encryption_compression OR NOT(nt_encryption_compression)) AND (nt_structured_type_method_specification_list OR NOT(nt_structured_type_method_specification_list))

```
Loading...
Completed Load: D:\TAF\TAF\Company_T\udt\udt.ttm
```

Figure 27. Model Representation of structured_type_create

6.5 RESULTS

A model and associated test driver schema produced 9,600 variations of the CREATE TYPE function, providing complete coverage of all function combinations. The traditional manual testing often would not attempt to cover all combinations. Following is an example of a few of the generated SQL language statements:

```
CREATE TYPE db9.type9 UNDER db8.type8 AS ( attr_def_list ) NOT FINAL ;

CREATE TYPE db1.type1 UNDER db2.type2 NOT FINAL ENCRYPTION INSTANCE
METHOD method_name ( params ) RETURNS data_type LANGUAGE C;

CREATE TYPE db1.type1 UNDER db2.type2 NOT FINAL ENCRYPTION OVERRIDING
CONSTRUCTOR METHOD method_name ( params ) RETURNS data_type ;

CREATE TYPE db9.type9 UNDER db8.type8 NOT FINAL ENCRYPTION ;

CREATE TYPE db9.type9 UNDER db8.type8 NOT FINAL ;
```

The TAF team also worked on some other functions, such as the CREATE TABLE function for the Partitioned Primary Index that used a similar pattern as shown in Figure 26. The team was able to use the same test driver schema to produce test scripts for the test environment.

There is a simple pattern to systematically model command line functions and language commands that often expand quickly into thousands of test cases. Manual testing often does not cover all these combinations. In addition, with the evolution required to support various other languages and environments, it is easier to modify the models and regenerate all the tests rather than to do the manual analysis and manual retesting of those features that have been impacted. In addition, once a modeler develops a common test driver schema for the target test environment, it can be reused as was done with the test script schema by this member.

The member also thought that the TAF approach captures feature-level requirements in addition to supporting automated test generation. The TAF model-based approach improves also on the typical script testing approach, where the requirements must be “internalized” by the test engineer (or developer in many cases) and then represented as a test script that is manually coded. TAF models separate the feature requirements (and logic) from the details of the test-scripting environment, which is generated automatically to carry out the tests. This provides requirement models that result in tests that cover application requirement threads more thoroughly than the traditional manual approach. This separates the test environment details from the model logic and should allow this member to evolve to a more advanced test environment, such as Java Database Connectivity (JDBC)[6] or Open Database Connectivity (ODBC), with automated results validation, which support SQL command execution. See more about the JDBC testing environment support for SQL and databases in Section 12.

[6] JDBC is a Java version supporting ODBC, a standard database access method developed by Microsoft Corporation.

This page intentionally left blank

7. CLIENT-SERVER WEB APPLICATION

7.1 PROBLEM

This member company builds many customer applications, some residing on older mainframes that use the IBM-style Customer Information Control System (CISC) through terminal emulation and some new client-server applications that interface with a database server through a web client. Most applications have little or no documented requirements to support the testing. In some small number of cases, user guides or help information may exist. The discovery of the requirements is left to the tester or a few experts. The company began investigating the use of Capture/Replay tools such as WinRunner to move toward more test automation. Although these tools do help in test automation, unlike model-based tools, they do not capture the requirements; rather they capture the sequence of keyboard and mouse operations that are required with a test scenario.

Many of this company's applications rely on a database too. To support testing, a database has to be populated with test data that can permit the execution of different test scenarios. Such a database is often referred to as the "gold" database. However, populating this type of database can be time-consuming, requires detailed understanding of the requirements, and usually is performed most quickly using some form of scripting or Job Control Language (JCL) in the case of this particular member application. For more details on how to address this problem see Section 12, which discusses model-driven dynamic test data generation for a database.

The TAF team worked with this member to develop a few different types of requirement models and test drivers that executed through a DynaComm CICS terminal emulation. The member wanted to build a model that could execute through a web-based client. The specific application discussed in this section is called the Strategic Problem Solving (SPS) system that was created through the WinRunner WebTest Capture/Play tool. The team selected this application because it is web-based and has WinRunner test scripts that provide the basis for understanding the requirements and previous test cases. The company also wanted to compare the TAF approach for generating tests from models to the tests manually created through WinRunner.

7.2 APPROACH

The TAF team worked with the company's lead test engineer to reverse engineer the requirements from existing WinRunner test scripts for the SPS application, which had a few test execution problems. The team also talked to two senior developers who understood the functionality of the SPS application and worked with staff to develop a model for a thread of requirements to demonstrate the feasibility. The modeled use case performed the Login to the

SPS system and then performed the Add Project function, which includes the following subfunctions to collect:

- General information about a problem
- Categories of the problem
- A free-text summary of the problem
- Participants involved in supporting and correcting the problem

The conceptual process and environment are similar to the earlier case studies, as reflected in Figure 28, but in this particular application, the interfaces to set the inputs were performed through the WinRunner scripting language that has API functions that can, for example, set the context to a particular window, push buttons, perform list menu selection, and edit data fields. Section 7.4 provides details.

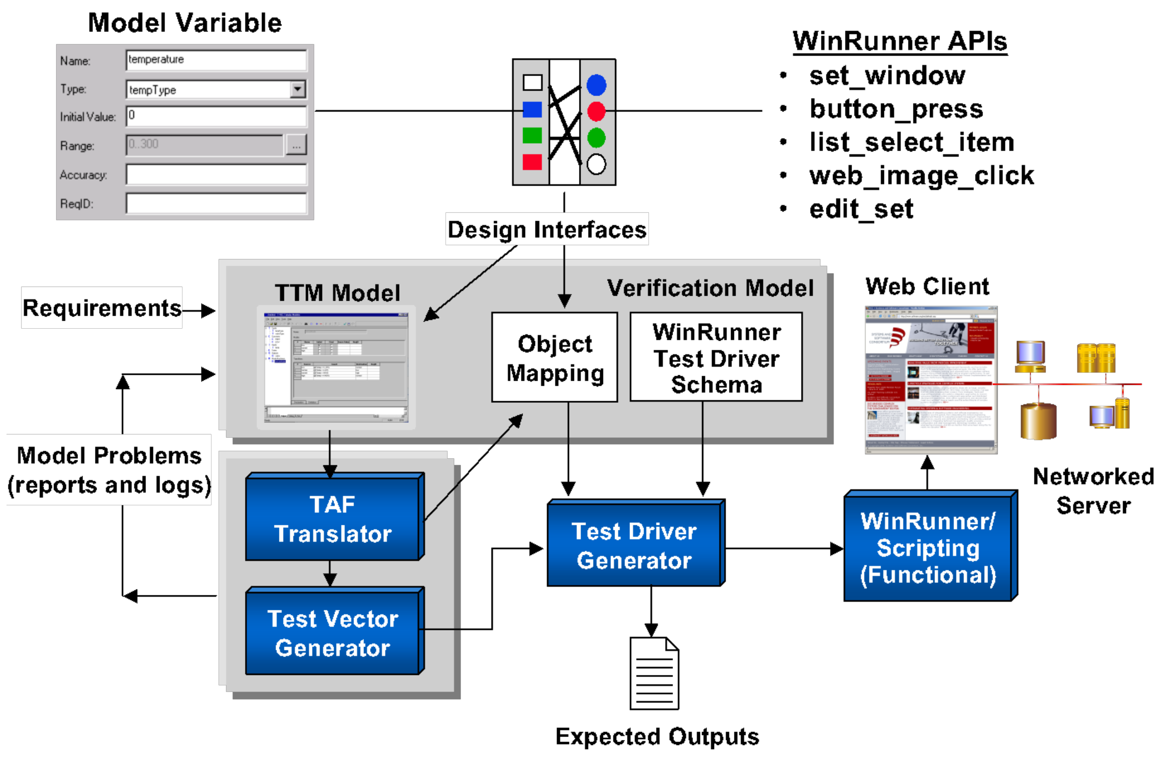


Figure 28. WinRunner Test Execution to Web Client

7.3 IMPLEMENTATION

The TAF team developed the model hand-in-hand with the analysis of the interfaces and WinRunner scripting language. WinRunner, like other capture/replay tools, provides a tool, called the GUI Spy, for identifying standard ActiveX and Java controls. It displays the properties of standard controls and the properties and methods of ActiveX and Java controls. These are used in the object mapping to relate the model variable to the controls of the web GUI application.

The team generated a test driver to support three subfunctions (General, Categories, and Free Text) and executed the generated script through WinRunner against a web browser (MS Internet Explore) interface to the application on a PC. The team extended and completed test driver mechanisms to support Strategic Problem Solving Login, and Participant information collection.

The team corrected the problems with the manually developed WinRunner script provided as initial input to this project. The model uses the more preferable Context Sensitive (as opposed to Analog) method for developing and executing WinRunner scripts.

7.4 KEY GUIDELINES

Capture/Replay tools orient the tester toward design for testing a thread through one application, and then a thread through another. This approach can be optimized using data tables to drive a single thread. However, if there are common features such as testing illegal key options for text field entries, different testers may spend significant time reentering the 50-plus illegal ASCII, HTML, and XML keys using the same sequence down various threads of the application. With TAF, which automates test design and generates scripts, a modeler would build the model that covers the combinations of keystrokes that can be reused for all types of text fields that require that type of testing. For more information on effective strategies, references, and guidelines for web and GUI testing see *Strategies for Web and GUI Testing* [Blackburn 2004].

7.5 RESULTS

The TAF approach captures requirements in addition to automated test generation support. These features improve on both the manual testing and WinRunner approaches, where the requirements must be “internalized” by the test engineer and then represented as a manual test or a WinRunner test script.

A TAF model of the requirements separates the application requirements (and logic) from the details of the test script, which is generated automatically to carry out the test. The requirement models result in tests that cover application threads more thoroughly than traditional manual approaches, and the requirement model is easier to review and assess and often provides the only documentation of the requirements.

The TAF modeling found problems with the manually created WinRunner scripts. Capture/playback tools are designed to record events either by context (object) or absolute position (analog). Recording based on context, such as the object name, prevents invalidating scripts when GUI controls are repositioned. Analog-based capture mechanisms store the absolute pixel position of each user event. A well-known disadvantage of basing the tests on analog-based screen location is that moving a button or menu a few pixels from the location that was captured can invalidate a script and require recapturing the session before the test can be reexecuted. During the analysis of the manually generated WinRunner script, the team determined a more appropriate approach for generating test scripts for an embedded table entry mechanism that is part of the Participant information. The object mapping fosters the development of test drivers that are predictable across many requirements. This is a fallout of the TAF process; the analysis support for test driver generation helps guide the use of common and standard test driver mechanisms that apply universally to applications, while reducing effort and maintenance.

The team identified types of web objects that did not support context control. The TAF process [Blackburn 2004] promotes design for testability, which fosters early communication and collaboration between the test engineers and developers. The team recommended that developers use standard sets of constructs (web objects) that allow for better/consistent mappings for context-sensitive-based testing automation, as described in the following example:

- In the SPS subfunction, where Participants are added, an embedded table is used to implement Adding and Modifying locations to the Participant information. Several issues related to the way the table addition mechanisms operate make it difficult to automate with WinRunner using the Context Sensitive mode. The fallback case relies on using physical screen locations, which is an approach that requires continual recapture of a session for any minor change of an application.

The TAF team was involved in this company's early stages of test automation evaluation. The other candidate test automation tools, such as WinRunner, were also in the initial stages of use. The TAF team did the following:

- Helped identify some key test automation requirements that were not apparent to some of the company's test engineers
- Helped illustrate how TAF automates the test design process through the generation of test cases, whereas, WinRunner still requires the tester to determine the different test cases
- Provided early support, which was later turned into the report "Understanding the Generations of Test Automation" [Blackburn 2003]
- Provided an important initial contribution by recommending a plan for performing an overall test automation evaluation, where a selected application could be used as a basis for a comparison of various test automation approaches
- Provided some key insights about organizational and development changes that would facilitate test automation

These insights and best practices were derived from performing pilot projects with other SSCI members over the past 8 years and are recorded throughout these case studies.

8. DISTRIBUTED BILLING SYSTEM

8.1 PROBLEM

The application, the PTN-II Usage Parser, processes billing records. As shown in Figure 29, there are several stages to the processing, which are done in parallel by separate distributed processes. Raw usage data is provided in control (.ctrl) and data (.data) file pairs. Through the COM process associated with the UNIX shell script run_com.sh.01, the raw data files are renamed and moved from the ptn2ready subdirectory to the local_ready subdirectory. Next, the PTN-II program is executed, and it moves and transforms properly formed control and data files into several other subdirectories, such as the ready subdirectory used by the billing system and the raw_ready subdirectory where inputs to the Usage Writer applications are placed.

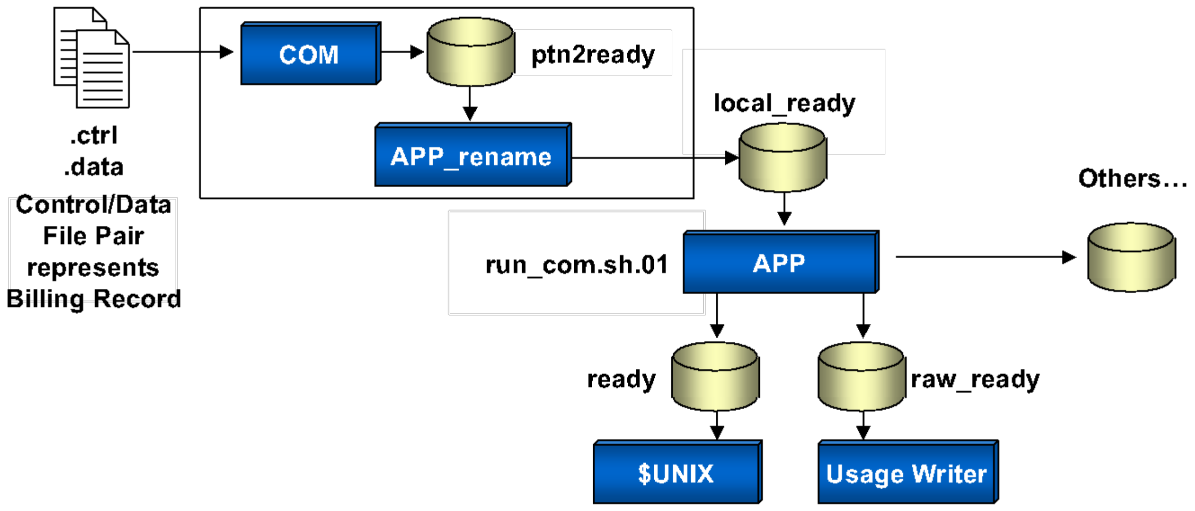


Figure 29. Conceptual Process Flow of Billing Record

The member company has a strong interest in doing more test automation because the current process is manually intensive, with a large number of combinations of information to process within each billing record. Continually adding new features makes testing these features and regression testing manually intensive. Often, testing is not performed in a comprehensive manner because of cost and schedule constraints. A few attempts had been made by this company to construct test scripts for automating the process, but no solution was ever completed. Therefore, there were no example test scripts on which to base the test driver generation process for supporting automated test execution.

The primary input is the Billing Specifications PTN-II Usage API document. It provides a good description of the interfaces, but the requirements are vague. Testing relies on domain experts, who understand the requirements and the way the system processes information through a multiprocess distributed application.

8.2 APPROACH

The objective was to develop models and test automation for the PTN-II Usage Parser. Figure 30 provides a high-level perspective of the process used to model, generate, and execute tests against the PTN-II (ptn2) parser application. The model was created by working backward from the interface definitions for the control and data records to ensure that all fields were complete. The customer specification provided good information about the representation of both records; however, the functional behavior (requirements), those associated with the actual values that each field of the record could take on, was mostly not documented.

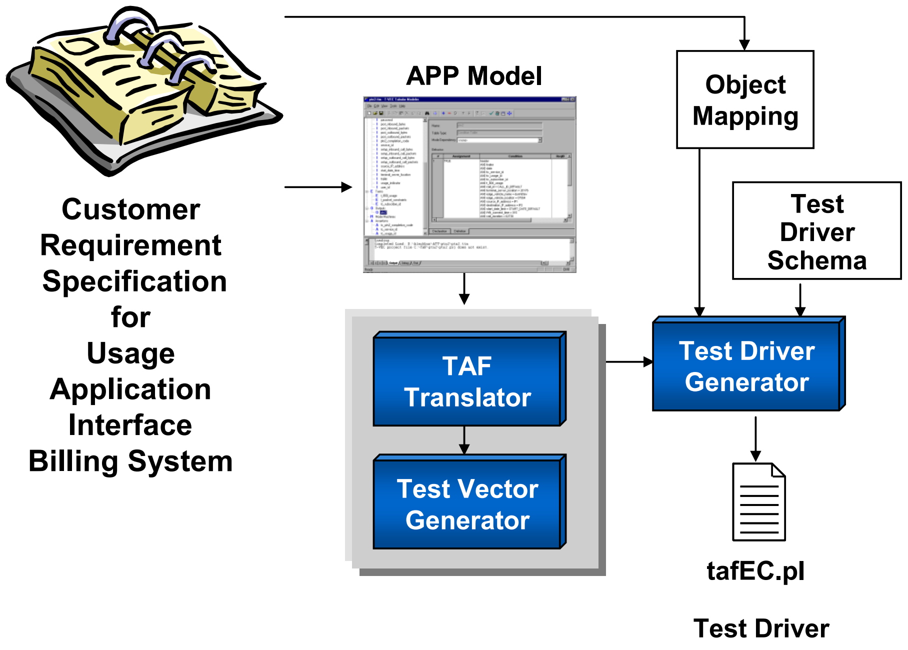


Figure 30. Distributed Billing System Modeling and Test Artifacts

8.3 IMPLEMENTATION

The TAF team worked with the company team, who described the requirement relationship between the input data and the resulting outputs. Because the API document provided excellent descriptions about the representation of the interface, it provided concrete information for the object mappings. The model was developed in the tabular modeling tool TTM. The model was translated, and test vectors were generated.

The TAF team helped the member company analyze the process by executing the COM and ptn2 programs to understand how to construct a test driver schema for automating the test execution. The team constructed object mappings that map logical model variables into fixed field representations as defined by the PTN-II API document. The test vectors, combined with the object mappings and test driver schema were input to the test driver generator, which produced a Perl language test driver. The test driver generated raw usage a control file with

UNIX checksum derived from the corresponding data record. The test driver includes the check to ensure that the transformed records are moved through the processing stages.

The test driver named tafEC.pl was created using TAF in a Windows 2000 environment. The PTN-II and other related applications run under UNIX. To execute the test driver, the modeler moved it to the networked UNIX environment using the File Transfer Protocol (FTP), as shown in Figure 31.

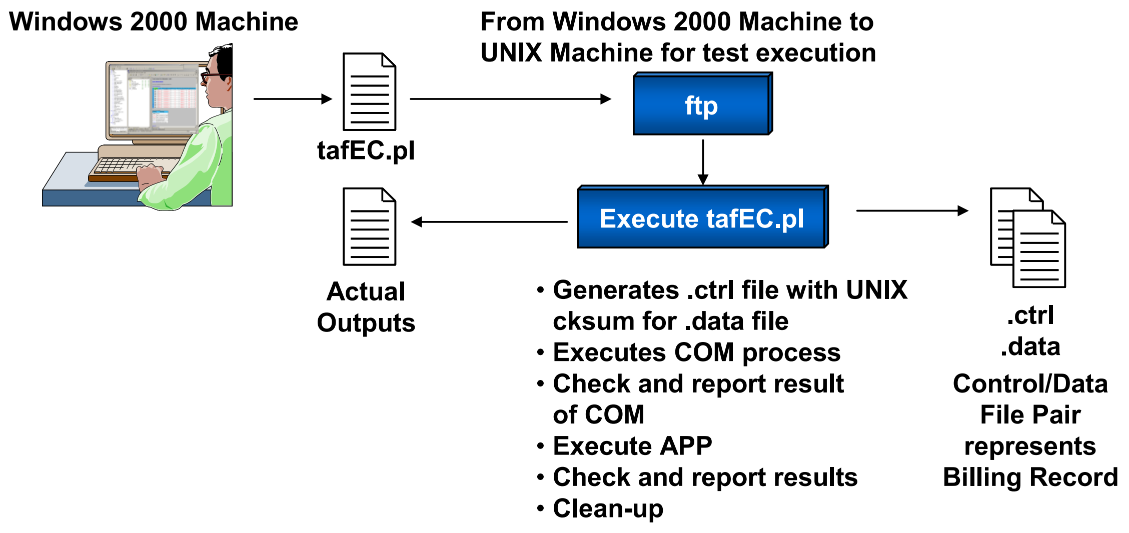

Windows 2000 Machine
From Windows 2000 Machine to UNIX Machine for test execution
tafEC.pl
ftp
Execute tafEC.pl
Actual Outputs
• Generates .ctrl file with UNIX cksum for .data file
• Executes COM process
• Check and report result of COM
• Execute APP
• Check and report results
• Clean-up
.ctrl
.data
Control/Data File Pair represents Billing Record

Figure 31. Distributed Billing Application Test Execution

A telnet window on the Windows machine was opened to the UNIX environment, and the test driver, tafEC.pl, written in Perl was then executed. The actual test data, which includes many test cases represented as combinations of control and data pairs, were embedded within in the Perl test driver. The Perl test driver looped through each test case, and for each test, a control and data file pair were generated with a unique UNIX time stamp prefixed to the headers. The generated control and data pair files were placed in the subdirectory of the COM process. As shown in Figure 29, the COM process is running in the background and monitoring new records that are placed in its subdirectory. The test driver checks to ensure that the files are renamed and properly moved from the ptn2ready subdirectory to the local_ready subdirectory. Next, the PTN2 tool is executed and the test driver checks to ensure that the files are moved from the local_ready subdirectory to the ready subdirectory. The test driver records the results of the checks as actual outputs that are then FTP'ed back to the Windows environment.

8.4 KEY GUIDELINES

This application reemphasizes the importance of interface-driven requirement modeling. This company's application interfaces with different companies or organizations that produce billing data, and as a result, their interfaces are well-defined. This definition was useful because every field of the billing record that needed to be modeled to create a valid billing record was known. Unfortunately, the requirements that describe the semantics for setting those data record fields to specific values were not as well-documented, but from a modeler's perspective, knowing that the interfaces exist makes it easier to ask questions until the requirements are extracted from those individuals or documents that can provide the behavioral information.

The distributed and multiprocessing nature of the non-Windows target systems serves to illustrate three more points:

1. It is possible to produce models and generate test cases in one environment, such as the Windows environment, and produce test drivers that execute in a different target environment. In working with member companies, it is common for the target environment not to be Windows.
2. The distributed processing environment uses time tags that are part of the control and data pairs to match those data pairs, as well as to uniquely distinguish different billing records. This application illustrated how embedded test data in the form of control and data pairs could be embedded within a program that is executed in the target environment to generate the actual test data in real time by using the UNIX system time function.
3. The outcomes associated with this particular application involved moving files, such as the control and data pairs, to different subdirectories and renaming the files in the process. In this case, the test driver served to monitor these events. However, if the real-time processing was significantly faster, it is possible that the test driver would not be able to monitor all the events. Therefore, an organization should make test logging part of the test infrastructure when possible. As discussed in [Blackburn 2004], the system designers should consider including other features that support testing, such as verbose output, event logging, assertions, resource monitoring, test points, and fault injection hooks [Pettichord 2002]. Verbose output and event logging can help trace bugs that are difficult to replicate. Assertions report incorrect assumptions in the application when it is running in debug mode. Test points and fault injection hooks support test execution [Blackburn 2004].

8.5 RESULTS

The objective to develop models and test automation for the PTN-II Usage Parser was achieved. Not only did the TAF team show that model-based testing applied to the company's application, but this was the first demonstration of test automation for this complex, distributed processing application. The modeling process also captured undocumented requirements that were known primarily by key developers.

TAF was demonstrated to apply to applications, where models developed in a Windows environment can produce test drivers that execute against applications hosted on a UNIX platform. The standard TAF process—used for embedded systems, mainframes, web-based applications, and database testing, for example—was used for PTN-II applications. This use means that this type of application is essentially the same as most other types of applications (from a TAF, model-based test automation perspective). The following list summarizes some of the benefits from requirement-based modeling:

- The TAF approach allows requirements to be captured in addition to supporting automated test generation. This improves on the typical manual testing approach, where the requirements must be "internalized" by the test engineer and then represented as a manual test.
- TAF models of the requirements separate the application requirements (and logic) from the details of the test script, which is generated automatically to carry out the test:

- This provides requirement models that result in tests that cover application requirement threads more thoroughly than a traditional manual approach.
- The model of requirements easier to review and assess.

Like many of the member companies, this member develops and maintains many applications, and the requirements are known by a few key people in the organization but often are sparsely documented. Modeling continues to be a useful process for documenting the requirements of key applications owned and used by member companies.

This page intentionally left blank

9. COMMAND AND CONTROL MONITORING SYSTEM

9.1 PROBLEM

This member company produces large, complex systems that often integrate with other large, complex systems. The application discussed in this case study performs a monitoring function for many elements of an onboard command and control system. This, like many of the other systems, continues to be evolved, and there are often many interactions among the various systems. The requirement specifications for this element of the system span hundreds of pages. There are many related requirements, but conflicting requirements are difficult to identify by inspection because they often are packaged in different volumes with many versions. In addition, the testing process is manual and performed on a requirement-by-requirement basis, which again leaves conflicting requirements unidentified.

This case study focuses on software-system integration functionality that is tested manually. The objectives were to demonstrate the capabilities of the TAF/T-VEC tools and method and help this company do the following:

- Assess how they can formalize requirements using models
- Assess where test automation is feasible
- Construct and demonstrate a test automation framework tailored to their system and environment
- Estimate the ROI over the existing manual test process
- Learn how to adopt technology and tailor processes

9.2 APPROACH

The names of the system elements have been changed to generic names to ensure anonymity for this company.

The primary system component for the pilot project is called the XYZ element. This element is part of a larger system call MASTER. There is requirement and interface documentation defined for this system, but important details, known by project personnel, were not in the documentation. The modeled requirements and associated tests primarily relate to messages received, processed, and transferred between various components and systems of the XYZ system. Currently, most of the testing is manual and performed against a target, although there is a simulator called XYZIPS that is used by the development organization to support test scripting using a language call Slang Script.

The XYZ system interfaces with several other large system elements of the XYZ, as reflected in Figure 32. From a high-level point of view, the MASTER system is composed of the XYZ, ABC, EFG, and LMN systems.

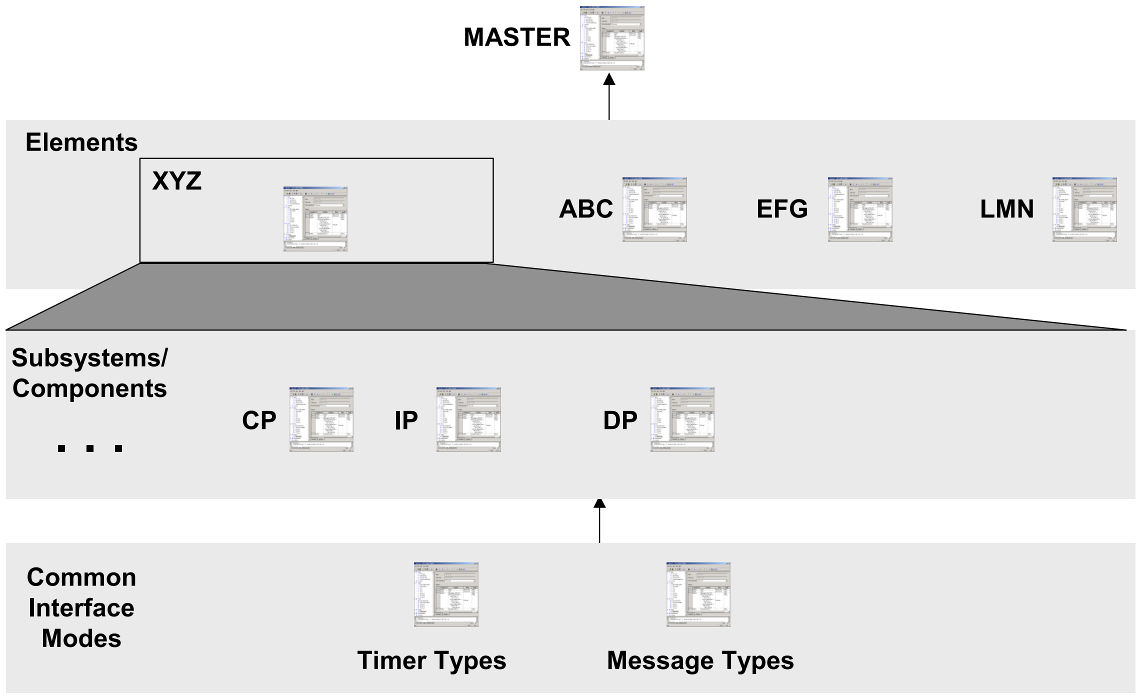


Figure 32. Model Hierarchy

Figure 33 provides a high-level perspective of the typical modeling process for the program. The objective is to use available requirement-related information, which comes in B-spec-like requirement documents, detailed interface specifications for the various messages, and some informal pictures that reflect analysis derived from requirement documentation and domain knowledge of the project engineers. Requirement models are specified in TTM, translated and T-VEC produces test vectors. The modeler, working with the requirement and design engineers, must correct requirement defects. Interface information is used to relate model variables to the actual system interfaces (API, message, etc.), and that supports automatic test driver generation of scripts that execute against a host, target or simulated system.

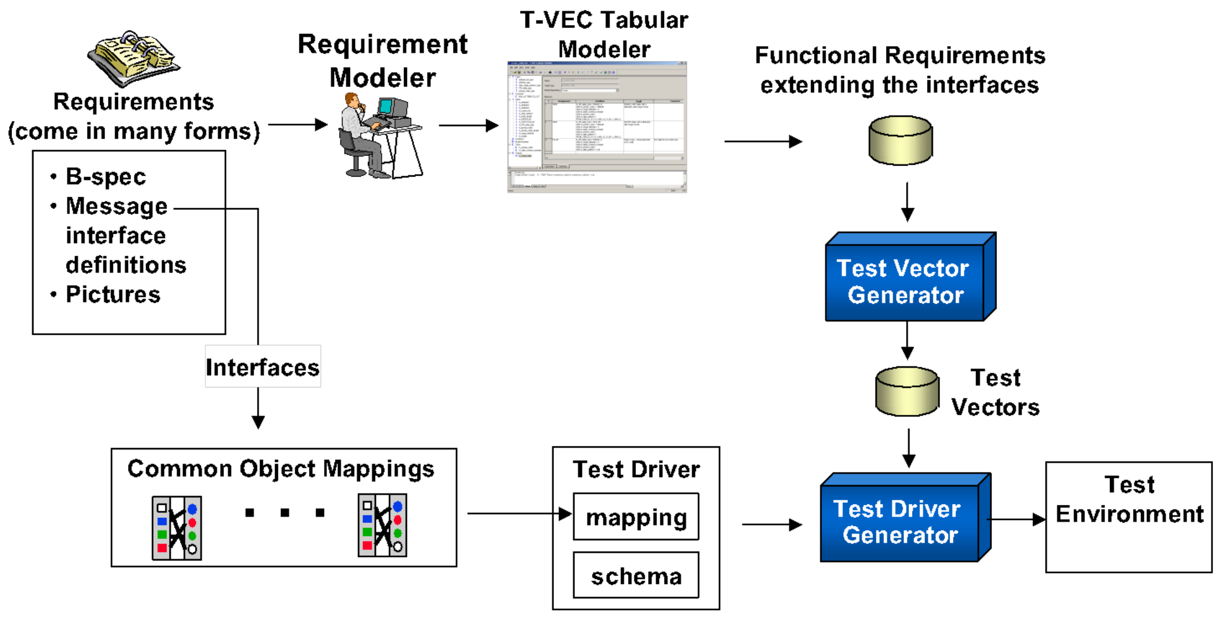


Figure 33. Model and Test Automation Overview

9.3 IMPLEMENTATION

The member company in conjunction with the TAF team carried out the TAF/T-VEC evaluation over three different phases. During the first phase, the TAF team was successful in developing models for various requirements and scenarios that were allocated to the XYZ system. The team worked from the documentation with knowledge from the key engineers to model 67 requirement threads resulting in 121 test vectors during the first 2 days of the pilot projects.

One of the model scenarios reflects the timed sequence interaction between two subsystems of the XYZ system (CP and IP), and the interaction to the XYZ. The company provided models to the team. An example textual specification follows:

- If in the Start state (mode), the incoming message is 100, the prompt is YES, and the IP is UP, then the system should transition to the Prompt state.
- If in the Start state, the incoming message 104, the prompt is YES, and the IP is DOWN, then the system should transition to Failure state.
- If in the Prompt state, the incoming message is 111, the prompt is YES, and prompt_timeout < PROMPT_TIMEOUT (has not timed-out), then the system should transition to the Ready state.

After developing some models for a number of these different scenarios, the team discovered that there were common models such as those reflected in Figure 32 called Timer Types and Message Type. The team was able to take advantage of the model include capabilities described in Section 2.5.2.3 to model the requirements once and then reuse the models for other related requirements. Section 14.4 describes this concept in more detail..

Between the first and second pilot visits, the key modeler developed a model that had 52 requirement threads with 77 unique test vectors. During the second phase, the team focused on developing test drivers for this model. The team developed a Slang Script that was executed through the XYZIPS simulator, and produced expected results that were observable in the data recorder logs produced by the XYZ system.

The team helped in the following ways:

- Identified some limitation in the XYZIPS simulator; it could execute only about 20 test vectors per test script. To overcome this limitation, the team modified the test driver schema to break large test driver script files into incremental files, each containing 15 test vectors.
- Identified some design for testability issues that need to be addressed with both XYZ system and XYZIPS to provide more general support for test automation.
- Talked with both the XYZIPS simulator developers and the XYZ system developer who indicated that future versions of both systems could be modified to support greater testability. For example, one key change would increase test automation by providing GUI events issued using messages rather than by a manual interaction with the system; this would permit the XYZIPS simulator to issue a program-generated message to the XYZ system without a human in the loop. Such commands could be generated by TAF.

The team next focused on the final stages of the test automation process, including automated test execution, results analysis, and test report generation. The team created a test results analysis program that extracted actual outputs from the raw data recorder information that is processed as a standard part of the XYZ process. These data recorder records provide the actual test outputs that are compared against the expected outputs produced by TAF/T-VEC. The comparison is recorded in an HTML test report file. Figure 34 shows the conceptual environment.

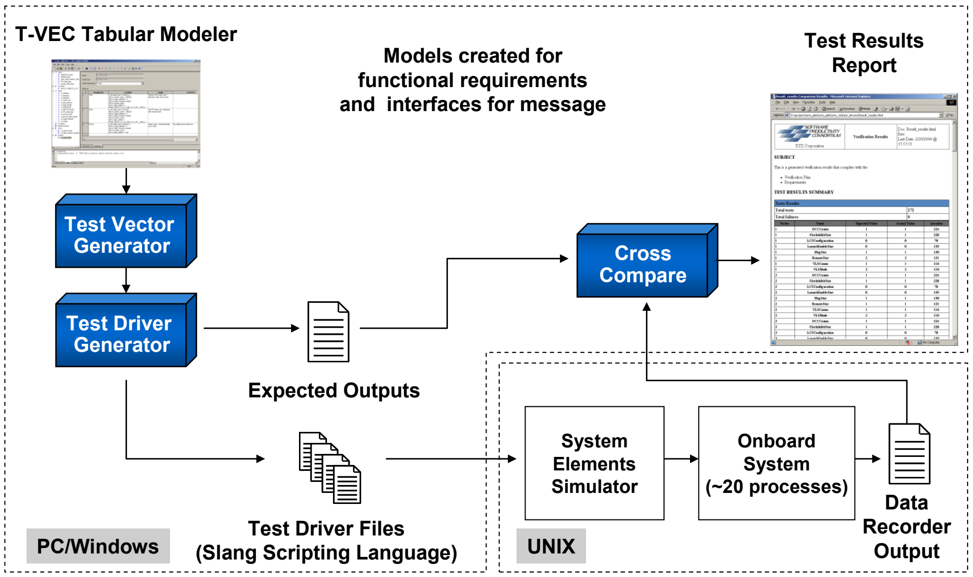

T-VEC Tabular Modeler
Models created for functional requirements and interfaces for message
Test Results Report
Test Vector Generator
Test Driver Generator
Cross Compare
Expected Outputs
System Elements Simulator
Onboard System (~20 processes)
Data Recorder Output
PC/Windows
Test Driver Files (Slang Scripting Language)
UNIX

Figure 34. Fully Automated Test Automation Process Flow for XYZ Testing

9.4 KEY GUIDELINES

9.4.1 ADOPT A TECHNOLOGY TRANSITION PLAN

The TAF team recommends that companies adopt a technology transition plan that grows the staff from the specialist developed in a prior project or pilot effort. The key modeler for this company demonstrated the skills of a model-based testing technologist, fulfilling the roles of both a modeler and test automation architect. He/she should be capable of carrying the recommended technology transition by leading one to six people in the use of the aforementioned process for a small set of requirements for the next release of some system baseline. The follow-on project then has additional team members to expand the team, where each person can mentor one to three additional people. When a company has a base of three to four project-experienced, model-based testing technologists, a larger group of 15 to 20 people can be trained to start a large project. SSCI provides generalized training using the *Model-Based Development and Automated Testing* course [Consortium 2003a] as a member service. In addition, we have tailored this course for different organizations, including this company, so that the course exercises use company-specific examples derived from the pilot project.

9.4.2 USE MODEL-BASED TESTING WITH OTHER TYPES OF TESTING

Model-based testing does not have to be all or nothing. Model-based testing can support a large percentage of the testing process, with a need to perform other types of testing. For example, there are two classes of messages processed by the XYZ system: solicited and unsolicited. According to COMPANY X resident expert, at least 70% of the messages sent to XYZ are unsolicited messages, sent by other XYZ elements that must be processed by XYZ without additional communication with the other elements. Figure 34 shows a complete end-to-end test infrastructure and process for handling unsolicited messages.

9.4.3 COLLABORATE WITH DEVELOPERS TO ADD TESTABILITY SUPPORT

For solicited messages, which are initiated by XYZ, usually through manual event, there is a need to enhance the testability of the current XYZ system in order to achieve complete test automation. External to the XYZ system, there is a need to have better controllability to simulate the external environment without manual intervention. It is also necessary to have some additional enhancements to the system for predictable observability of the test outputs. The team discussed the need, with the developer, to add a programmatic interface to the system that would allow external programs to stimulate system events such as solicited messages. The lead developer said it would be feasible and relatively inexpensive to implement the recommended programmatic interface in the next release.

9.4.4 USE SIMULATION FOR EARLY AND CONTINUOUS TESTING

Coincidently, the XYZIPS simulation team was planning to develop a new simulator, and they were open to taking new requirements. Although there were a few limitations in the old XYZIPS simulator, the TAF team's use of the simulator was valuable in providing requirements to further enhance their simulator. We were able to overcome the XYZIPS limitation where the system cannot execute a Slang Scripts with more than about 20 tests. We modified the schema to produce multiple test driver files containing only 15 test cases each. We added a parameter that can be used to configure the test driver schema to put any number of test cases in a single test driver file.

9.4.5 Leverage System Internals for Analyzing Actual Test Outputs

Design for testability is key to test automation because it is important to be able to programmatically set inputs or initiate events, as well as obtain outputs that reflect the system behavior. The XYZ system receives thousands of messages but does not necessarily send out a response for each message; however, it does have a data recorder (i.e., internal logging) that collects all message inputs and associated process outputs. This member company lead determined that the raw data recorder file could be processed to extract actual outputs for automated test results analysis. The TAF team developed an XYZ test results comparison program in Perl to extract the actual outputs from the data record file and compare them with the expected outputs produced by the test vector generator, while producing an HTML test results report.

9.5 RESULTS

The pilot effort was successful in demonstrating an automated, end-to-end test process, as shown in Figure 34, that could be an order of magnitude more comprehensive than manual testing with 50% less time and cost. Prior to the pilot project, nearly all system and integration testing was a completely manual process. The pilot demonstration resulted in a few hundred test cases that represented several thousands of cases that are normally executed manually when the system capability is first developed, but then manually executed many times for each time the system is regression tested.

Using the TTM, various requirements were modeled from XYZ requirements, interface specifications, and domain knowledge of resident experts. The test design process was fully automated using automatic test vector generation from the models and test driver generation capabilities to produce test scripts that automatically execute through the XYZIPS simulator. XYZIPS interfaces to XYZ by sending messages in a manner equivalent to all other MASTER system elements. XYZ processes each test message, and a data recorder process within XYZ captures all internal processing details, which are output to a data recorder file. The automated test results analysis is performed by comparing the expected outputs, which are derived from the model and test vector generator, with the actual outputs that are extracted from the data recorder information produced by XYZ.

The pilot project results are significant because the demonstrations were conducted using XYZ message requirements that represent 70% (or possibly more) of XYZ message processing, for which tests are currently performed manually. To carry out additional testing using this TAF/T-VEC process will require some minor changes to XYZIPS and XYZ. The pilot demonstrated the feasibility of using company technical staff to develop models. With just the initial 2-day pilot project training, the key member company modeler (primarily systems tester, not program developer) developed additional models for different types of messages and extended the test driver schemas to support Slang Script generation. The pilot demonstrated the feasibility to generate test drivers automatically in the script language that is interpreted by the XYZIPS simulator when it communicates to XYZ. The test results report is produced in HTML format through a cross-comparison program that compares the expected outputs with the actual outputs stored with the raw data recorder output of the XYZ system. The test infrastructure for test driver generation and automated test results analysis and report generation are sufficiently robust to support follow-on XYZ model-based testing tasks.

There are many other intangibles ROI benefits. This process and the supporting test infrastructure used to support this pilot demonstration was 80% to 90% complete and relatively stable to support all follow-on testing. In addition, the team identified several requirements for the testing infrastructure that could further automate the process or change the underlying process for the organization. For example: Once an automated test suite exists, it can be run each time a build of the system occurs, allowing developers to identify bugs earlier in the process and making it easier to understand the specific changes that introduced a defect into the system rather than waiting weeks or months before manual testing is performed.

Another important benefit that was observed during the requirement modeling process is that important requirement details often are not reflected in the requirement documents. Once the model is developed, these important details are captured (from domain experts). Related requirements that often span different pages in a requirement document are captured in the same model. Companies often find that the captured models are the most valuable asset because they not only specify requirements in a nonambiguous manner, but they are the source of the tests that can be generated systematically to provide complete test coverage from the requirements.

The first use of this technology requires some learning, but the first use by other companies indicates that even in the first use there is a significant increase in test coverage in less time than with existing manual processes. The key ROI gains should be obtained with each addition regression testing session that occurs. For this company, the time required for regression testing is essentially the same as it is to test the first time. With this automated process, the time to perform regression testing is easily less than 50% of the original time and cost and can be as little as 10%.

The TAF team executed a complete demonstration of the process to the management team, showing the new process successfully applied to their target application.

This page intentionally left blank

10. TIME CARD LOGIC PROCESSING

10.1 PROBLEM

This company produces both internal and external applications. They were interested in advancing their testing processes and selected one component application of the supporting infrastructure to use in an evaluation of the TAF/T-VEC tools. The application is a component of a time card processing system. The requirements are not well documented, but are understood by key staff. They current process relies on the manual creation of data and manual analysis of the outputs. They are interested in how to better automate testing.

10.2 APPROACH

The approach followed the common pattern for model development as shown in Figure 35. The company team sketched the rules for the timecard logic on a whiteboard and then modeled them. Test vectors were generated. Object mappings were created to describe the relationship between the modeled variables and detailed representation of the fields of the records output to the time_cards_subtotal.txt file.

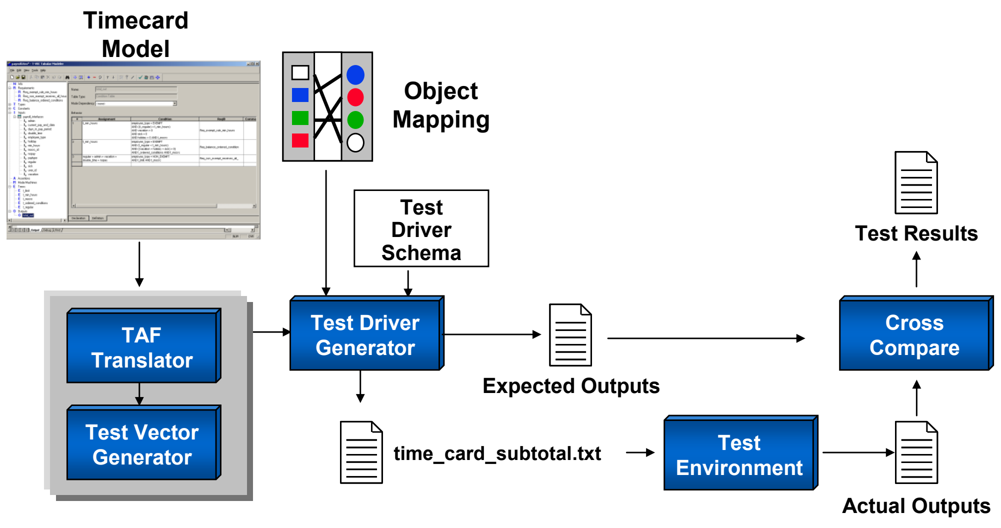
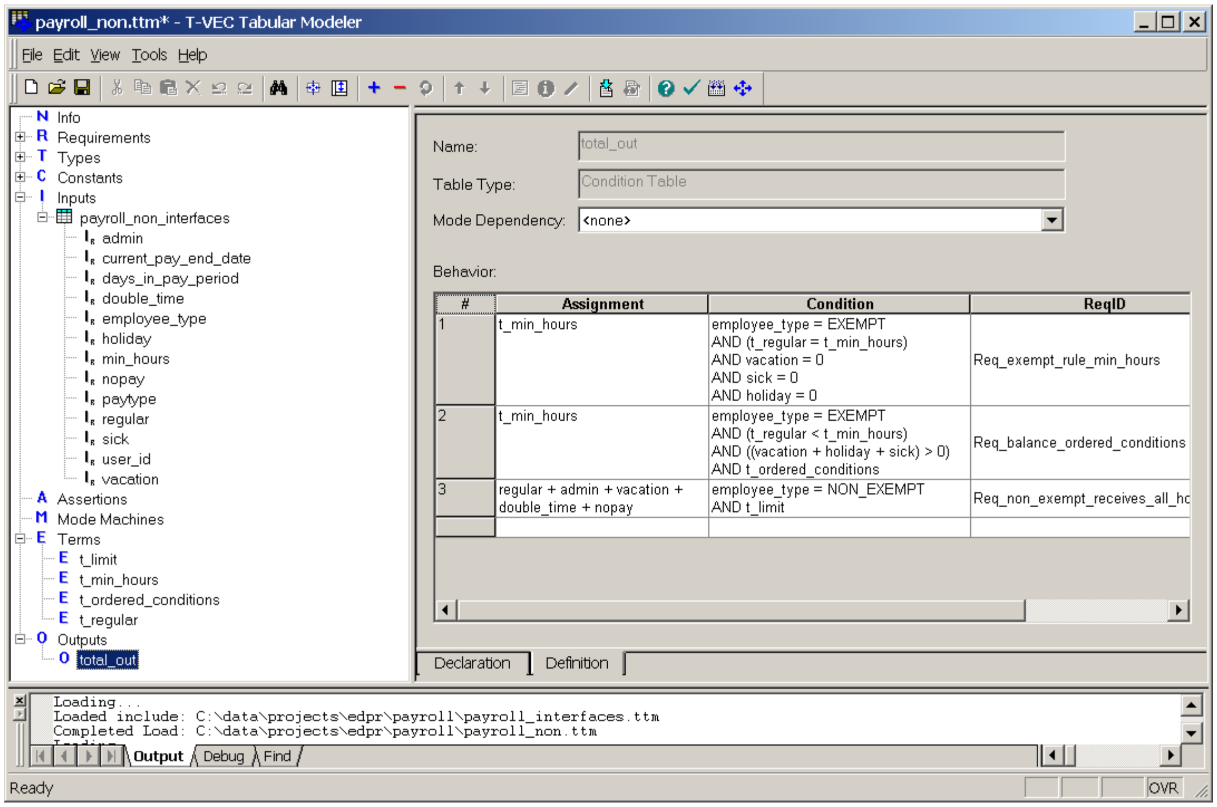


Figure 35. Time Card Logic Modeling Process

10.3 IMPLEMENTATION

The model, shown in Figure 36, had one output that was associated with information related to the rules for time cards for exempt and non-exempt employees, in addition to a time card ordering requirement that describes a rule to ensure that each time card has a proper number of hours. Section 10.4 describes details of the model and some new concepts. After translating the model and generating test vectors, 110 unique test vectors were produced using a translation option called “inlining.” Lastly, the 110 test vectors resulted in 265 time card records. Section 10.4 also discusses the inlining concept and the time card records.



Name: total_out

Table Type: Condition Table

Mode Dependency: <none>

Behavior:

#	Assignment	Condition	ReqID
1	t_min_hours	employee_type = EXEMPT AND (t_regular = t_min_hours) AND vacation = 0 AND sick = 0 AND holiday = 0	Req_exempt_rule_min_hours
2	t_min_hours	employee_type = EXEMPT AND (t_regular < t_min_hours) AND ((vacation + holiday + sick) > 0) AND t_ordered_conditions	Req_balance_ordered_conditions
3	regular + admin + vacation + double_time + nopay	employee_type = NON_EXEMPT AND t_limit	Req_non_exempt_receives_all_hc

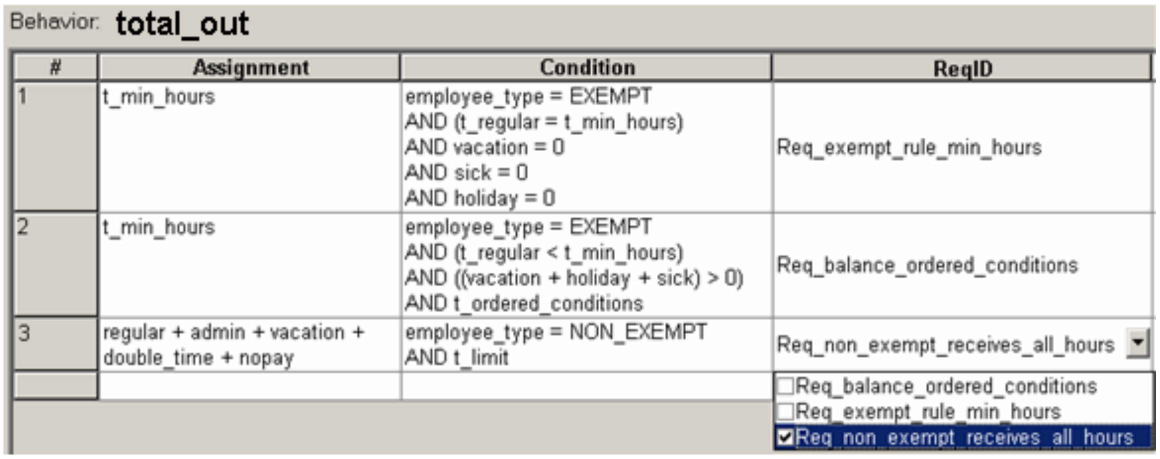
```
Loading...
Loaded include: C:\data\projects\edpr\payroll\payroll_interfaces.ttm
Completed Load: C:\data\projects\edpr\payroll\payroll_non.ttm
```

Figure 36. Time Card Logic Model

10.4 KEY GUIDELINES

10.4.1 MODELING PRACTICES

This section discusses several recommend practices, including the use of naming conventions, terms that can be reused throughout the model, constants, and traceability links to requirements. Figure 37 shows an example of the output condition table for the output variable total_out.



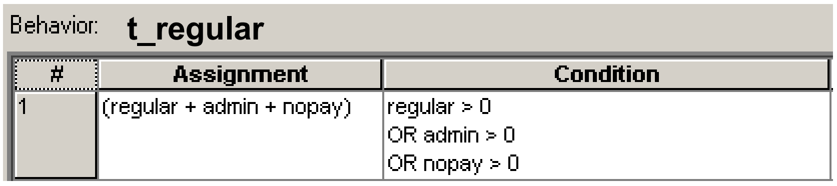
Behavior: **total_out**

#	Assignment	Condition	ReqID
1	t_min_hours	employee_type = EXEMPT AND (t_regular = t_min_hours) AND vacation = 0 AND sick = 0 AND holiday = 0	Req_exempt_rule_min_hours
2	t_min_hours	employee_type = EXEMPT AND (t_regular < t_min_hours) AND ((vacation + holiday + sick) > 0) AND t_ordered_conditions	Req_balance_ordered_conditions
3	regular + admin + vacation + double_time + nopay	employee_type = NON_EXEMPT AND t_limit	Req_non_exempt_receives_all_hours
			☐Req_balance_ordered_conditions ☐Req_exempt_rule_min_hours ☑Req_non_exempt_receives_all_hours

Figure 37. Time Condition Table for total_out

The TAF team added the time card processing requirements directly into the TTM tool, however requirements can be linked from the DOORS requirement management tool as discussed in Section 2.5.2. Traceability is easiest to follow, if there are a few requirement statements associated with one requirement identifier. One or more requirement identifiers can be linked to one row of a condition table as shown in Figure 37. Consider row 1 for example, the conditions when an exempt (i.e., salaried) employee satisfies the constraint where the minimum hours (t_min_hours) are equal to the expected regular number (t_regular) of hours for that pay period is associated with the requirement identifier (ReqID) Req_exempt_rule_min_hour as shown in Figure 37. Row 2 defines the conditions for balancing the hours for an exempt employee, when the minimum hours worked is not equal to the regular expected hours for a particular pay period; this is done by balancing the vacation, holiday, and sick hours. Row 3 defines the conditions for defining the total hours for a non-exempt (i.e., hourly) employee.

It is a good practice to define common conditions or expressions that are referenced by other tables in a term table. For example, the term t_regular defines the condition and expression for the regular number of hours, as shown in Figure 38. When the regular, admin, or nopay hours are greater than 0, the term t_regular is equal to the sum of those hours. Also note that a prefix of “t_” is used before each term variable. This makes it easier to identify named expressions that are terms rather than input variables or constant.

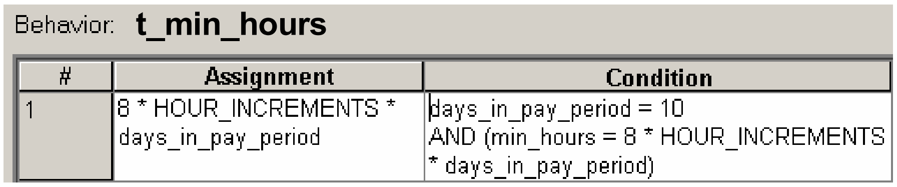


Behavior: **t_regular**

#	Assignment	Condition
1	(regular + admin + nopay)	regular > 0 OR admin > 0 OR nopay > 0

Figure 38. Term Table for Regular Hours

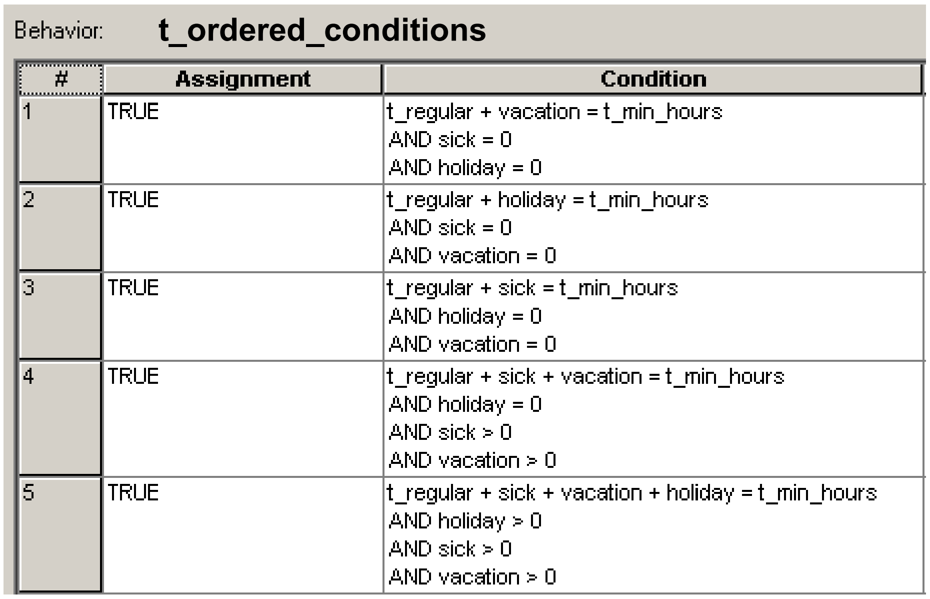
The term table for the minimum hours (t_min_hours) is shown in Figure 39. This term defines the number of hour increments in a pay period. A pay period can be 10 days, where each day consists of 8 hours. It is also a good practice to use upper-case characters for constants such as HOUR_INCREMENTS. This makes it easy to distinguish constants from input variables and terms. As shown in Figure 37, it is recommended also that enumerated constants such as EXEMPT be defined in upper-case characters.


Behavior: **t_min_hours**

#	Assignment	Condition
1	8 * HOUR_INCREMENTS * days_in_pay_period	days_in_pay_period = 10 AND (min_hours = 8 * HOUR_INCREMENTS * days_in_pay_period)

Figure 39. Term Table for Minimum Hours

The term table for the scenario where the regular hours worked by an exempt employee does not meet the minimum required hours for a pay period requires the employee to use either sick, holiday, or vacation hours. The rules for covering those particular conditions are shown in Figure 40. Each row is logically an exclusive OR'd situation from each of the other rows in the table. Note, that the assignment for each row is simply TRUE, which means that this condition could have been modeled as an assertion because at least one condition must be TRUE (i.e., there are no FALSE conditions for this particular table). For example, row 1 defines the case when the employee must use vacation hours when the holiday and sick hours are 0, while row 4 forces the t_min_hours to be equal to the sum of the t_regular, sick and vacation hours because holiday hours equals 0.


Behavior: **t_ordered_conditions**

#	Assignment	Condition
1	TRUE	t_regular + vacation = t_min_hours AND sick = 0 AND holiday = 0
2	TRUE	t_regular + holiday = t_min_hours AND sick = 0 AND vacation = 0
3	TRUE	t_regular + sick = t_min_hours AND holiday = 0 AND vacation = 0
4	TRUE	t_regular + sick + vacation = t_min_hours AND holiday = 0 AND sick > 0 AND vacation > 0
5	TRUE	t_regular + sick + vacation + holiday = t_min_hours AND holiday > 0 AND sick > 0 AND vacation > 0

Figure 40. Term Table for Ordered Conditions

10.4.2 Test Vector Generation From Hierarchical Models

Each table of the model is translated into a T-VEC subsystem. A T-VEC subsystem has declarations of types, constants and variables, and then contains a section that represents the assignment part of the table and associated constraints that represent the information contained in the column labeled Condition. For details on the T-VEC linear form, refer to the T-VEC Language Reference Manual, which comes with the tool installation. T-VEC generates test vectors for each subsystem in the hierarchy, as shown Figure 41. By default a translated model is compiled into a set of pre and postcondition pairs. A precondition (Domain Convergence Path [DCP]) defines constraints on inputs associated with conditions in a model table. A postcondition defines the output as a function of the constrained inputs and is associated with the assignment of a model table.

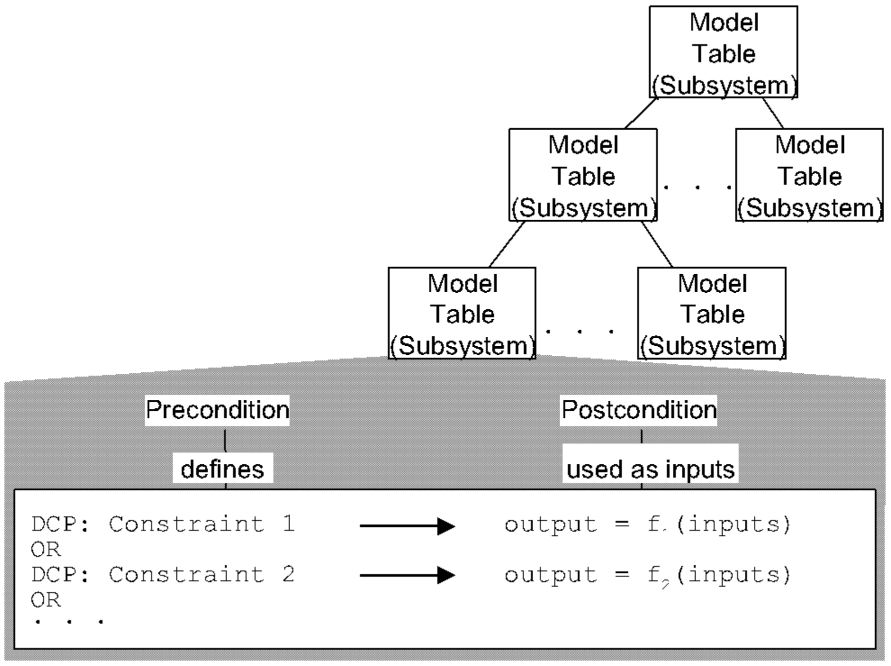


Figure 41. Hierarchical Model and Translated Representation

By default, the translation converts outputs tables as well as term tables into subsystems that are related hierarchically. T-VEC subsystems have properties, which are stored in the T-VEC project file. Term tables, by default, have a compile-only property that is enabled. Test vectors are not produced for subsystems with the compile-only property. Normally, it is not necessary to produce test vectors for a term table because there is no output associated with the term variable.

To generate test vectors, T-VEC first compiles the translated model. The compilation process performs three primary functions, as follows:

- Converts the specification into a form, called system knowledge, that is appropriate for processing by the test vector generator; this includes detailed information about the data types, variables, and the behavioral specifications that are referred to as DCPs.
- Checks that interface references to other hierarchically dependent subsystems are correct syntactically. Unless there is an model error that is ignored prior to translation, the translator should produce the proper interface references to hierarchically related subsystems.
- Constructs test justification paths that are used in test coverage analysis to assess the extent of the model-based test coverage.

The compilation process ensures that the target specification is syntactically correct and that references to other “external” system specifications are semantically correct with respect to the input/output type definitions of the system being referenced. Once the compilation process completes, the results are stored in a system knowledge file.

TAF supports hierarchical relationships. In generating test vectors for a hierarchy of models, as represented in Figure 42, 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.

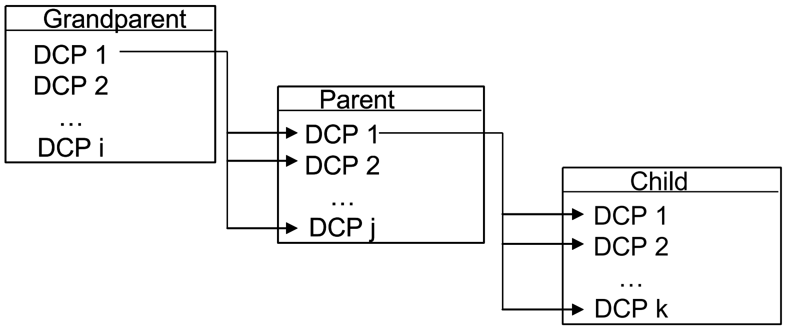
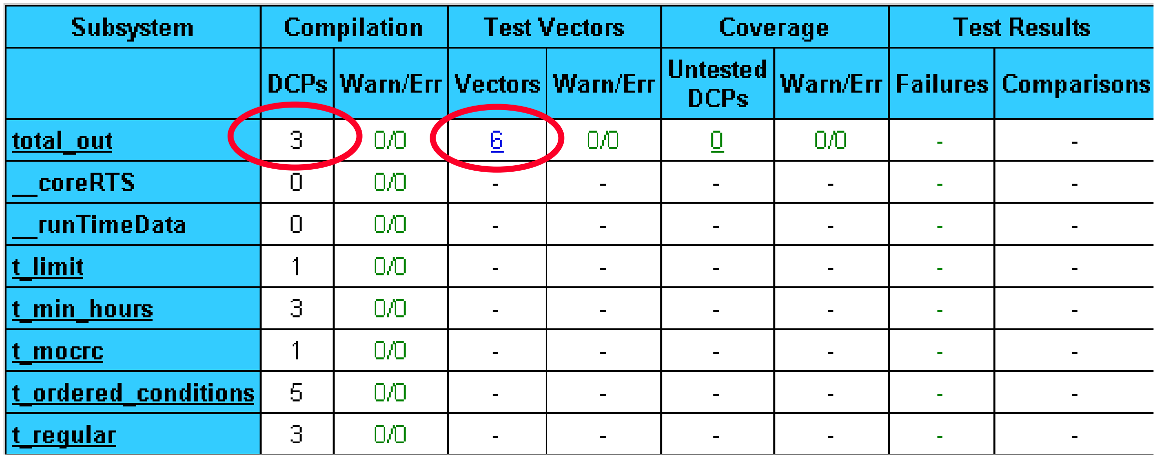


Figure 42. Hierarchical Subsystem Relationships

The project status, shown in Figure 43, reflects that there are six test vectors generated from the three DCPs of the parent-level subsystem total_out. Each row of the total_out table, shown in Figure 37 corresponds to one DCP. The T-VEC test generation system uses a test selection heuristic based on domain testing theory [White 1980] 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 [Tsai 1990].



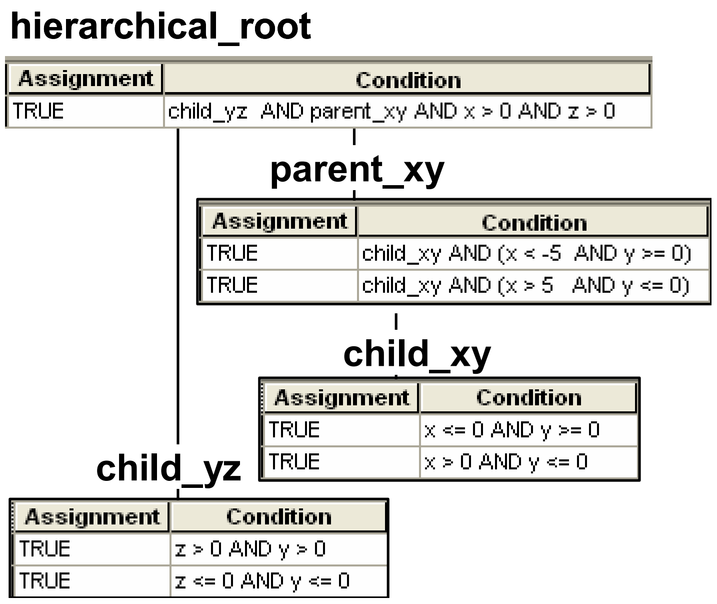
Subsystem	Compilation		Test Vectors		Coverage		Test Results	
	DCPs	Warn/Err	Vectors	Warn/Err	Untested DCPs	Warn/Err	Failures	Comparisons
total_out	3	0/0	6	0/0	0	0/0	-	-
__coreRTS	0	0/0	-	-	-	-	-	-
__runTimeData	0	0/0	-	-	-	-	-	-
t_limit	1	0/0	-	-	-	-	-	-
t_min_hours	3	0/0	-	-	-	-	-	-
t_mocrc	1	0/0	-	-	-	-	-	-
t_ordered_conditions	5	0/0	-	-	-	-	-	-
t_regular	3	0/0	-	-	-	-	-	-

Figure 43. Project Status for Time Card Model

10.4.3 Model Defects

A reference from an upper-level subsystem to a lower-level subsystem must be satisfiable by at least one DCP in the lower-level subsystem. Constraints of the upper-level subsystems are applied to the parent DCP before the references to any child subsystem are considered. Each DCP of the child is ANDed with the DCP of the parent until the combination is satisfied. If there is no DCP thread from a higher-level subsystem to a lower-level subsystem, this proves that there is no input space associated with the model (i.e., the input space for the DCP is null). When generating test vectors, the inputs are selected from the inputs, but if the input space is null, no tests can be selected; this is an invalid specification within the model.

The example in Figure 44 represents a trivial model with four condition tables. This simplified model has a seeded defect to illustrate the model traceability links from a model report to the model. The tables have dependency relationships to illustrate the use of model traceability. Each row of each table in the transformed model has a one-to-one correspondence with a DCP thread. The highest-level subsystem, hierarchical_root has one DCP that references child_yz, and parent_xy, each with two DCP threads. Parent_xy references child_xy, which also has two DCP threads.



hierarchical_root

Assignment	Condition
TRUE	child_yz AND parent_xy AND x > 0 AND z > 0

parent_xy

Assignment	Condition
TRUE	child_xy AND (x < -5 AND y >= 0)
TRUE	child_xy AND (x > 5 AND y <= 0)

child_xy

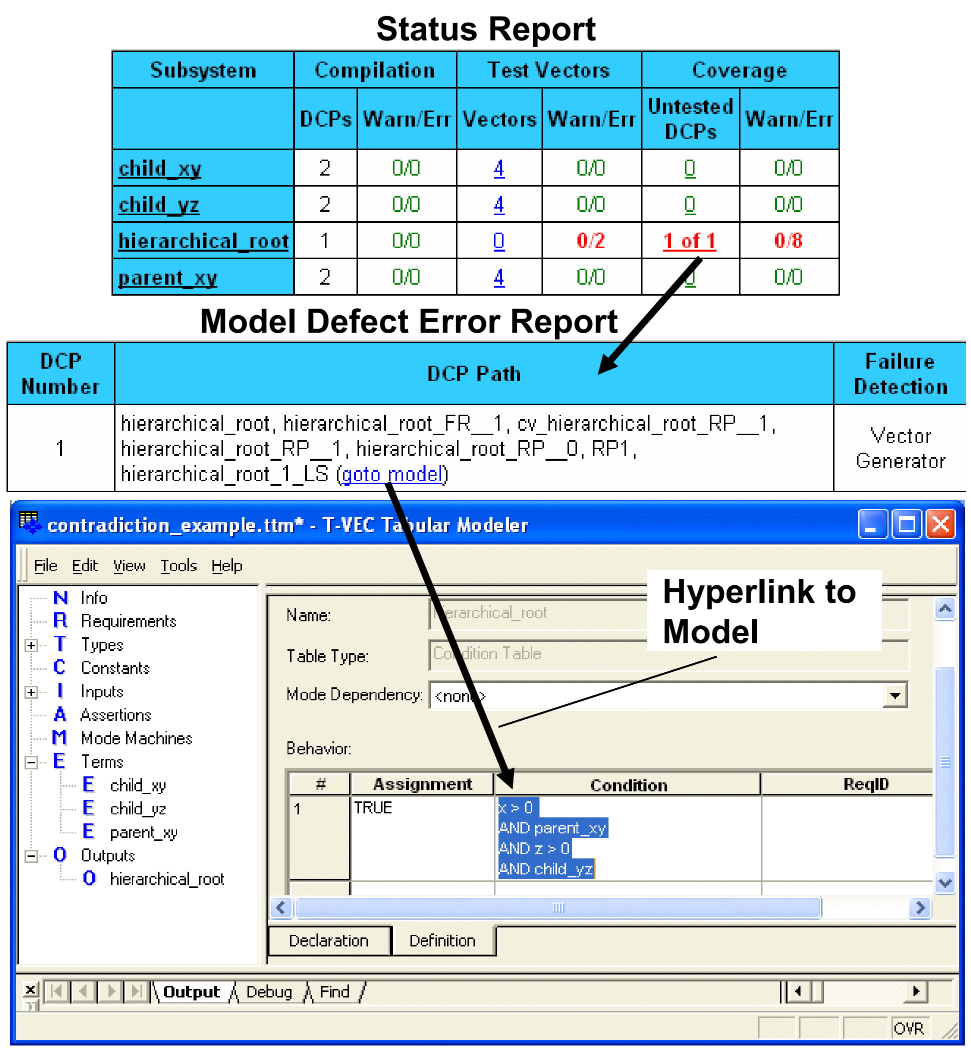
Assignment	Condition
TRUE	x <= 0 AND y >= 0
TRUE	x > 0 AND y <= 0

child_yz

Assignment	Condition
TRUE	z > 0 AND y > 0
TRUE	z <= 0 AND y <= 0

Figure 44. Hierarchical TTM Model

Figure 45 shows the traceability links from the status and error reports to the likely source of the model error. The status report provides a summary for each subsystem, including the number of DCPs derived during the compilation process of the model. The summary report provides the number of test vectors and the number of model coverage errors. Hyperlinks from the project status report link to other reports, including the model defect error report, which is produced for each DCP that has a defect. A hyperlink from the model error report traces back to the model specification that is the likely source of the problem.



Status Report

Subsystem	Compilation		Test Vectors		Coverage	
	DCPs	Warn/Err	Vectors	Warn/Err	Untested DCPs	Warn/Err
child_xy	2	0/0	4	0/0	0	0/0
child_yz	2	0/0	4	0/0	0	0/0
hierarchical_root	1	0/0	0	0/2	1 of 1	0/8
parent_xy	2	0/0	4	0/0	0	0/0

Model Defect Error Report

DCP Number	DCP Path	Failure Detection
1	hierarchical_root, hierarchical_root_FR__1, cv_hierarchical_root_RP__1, hierarchical_root_RP__1, hierarchical_root_RP__0, RP1, hierarchical_root_1_LS (goto model)	Vector Generator

Name: hierarchical_root

Table Type: Condition Table

Mode Dependency: <none>

Behavior:

#	Assignment	Condition	ReqID
1	TRUE	x > 0 AND parent_xy AND z > 0 AND child_yz	

Figure 45. Model Defect Traceability to TTM

The defect exists because there is no combination of DCP threads through the lower-level subsystems that permit both x and z to be greater than 0 when the output (i.e., assignment) of hierarchical_root must be TRUE. The model child_2_xy requires y <= 0 when x > 0, but child_2_yz requires y > 0 when z > 0. Thus, a contradiction exists between the logic of hierarchical_root and logic across two dependent subsystems. This is a trivial example, but this type of situation is commonly called a feature interaction problem. Section 11 discusses this issue in more detail.

10.4.4 Test Vector Generation From Inlined Tables

T-VEC subsystems generated from inlined subsystems contain the logic for inlined subsystems and are tested within the parent. Conceptually, the translator forces the DCPs of the lower-level subsystems to be included in the higher-level subsystem as shown in Figure 46. Inlining forces every combination of lower-level subsystem to be AND'ed with every lower-level subsystem. This can result in a large number of combinations but generates the tests that support comprehensive testing from the higher-level system. This is sometimes needed to tests the implementation associated with lower-level subsystems that do not have accessible code entry points, and therefore the testing must be performed through the Parent's (or Grandparent's) code entry points.

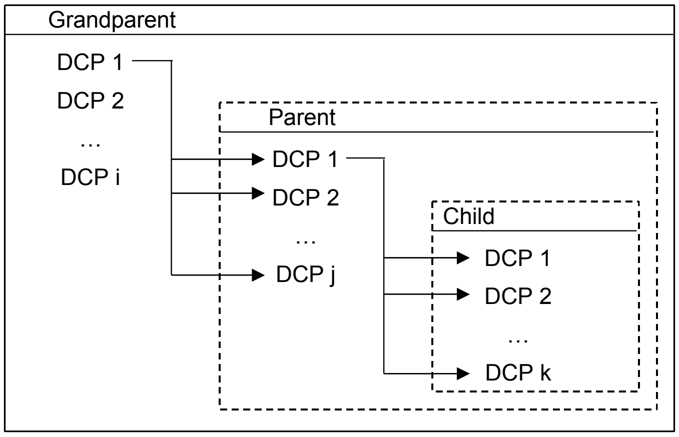
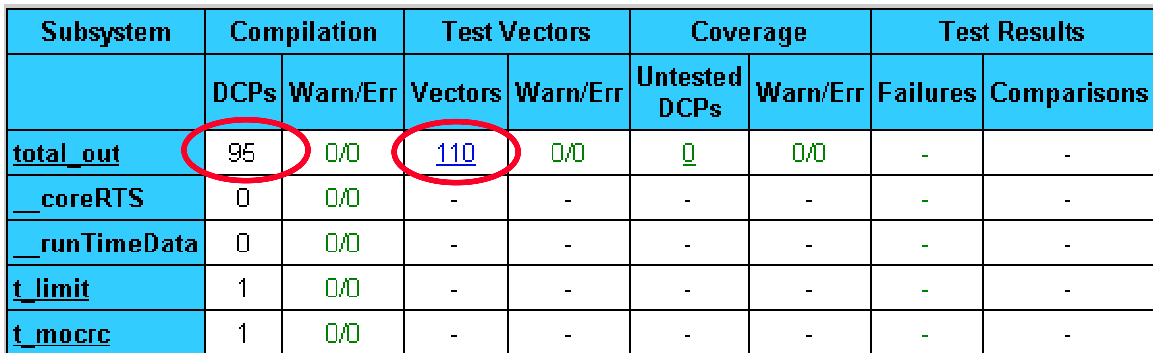


Figure 46. Inlined Subsystems Included in Higher-Level Subsystems

The project status, shown in Figure 47, reflects that there are 110 test vectors generated from the 95 DCPs of the parent-level subsystem total_out once the term tables were inlined. The resulting test vectors exercise all combinations of the time card processing logic.



Subsystem	Compilation		Test Vectors		Coverage		Test Results	
	DCPs	Warn/Err	Vectors	Warn/Err	Untested DCPs	Warn/Err	Failures	Comparisons
total_out	95	0/0	110	0/0	0	0/0	-	-
__coreRTS	0	0/0	-	-	-	-	-	-
__runTimeData	0	0/0	-	-	-	-	-	-
t_limit	1	0/0	-	-	-	-	-	-
t_mocrc	1	0/0	-	-	-	-	-	-

Figure 47. Project Status for Time Card Model With Inlined Term Tables

10.5 RESULTS

The modeling process illustrated how textual requirement definitions or nonexistent requirements can be improved. The modeling process also helped capture the key requirements that were most completely understood by the most senior member of the team. The generated tests were significantly more comprehensive than the typical test sets that are performed manually. The TAF team produced test inputs from the model in a format that required two files: a time card status file, and a time card subtotals format. The team then changed the test driver to produce a consolidated test input in one file, as reflected in Figure 35. These changes were done without any modification to the model. Rather, a few changes were made to the object mapping to consolidate the generated test information into one file that was loaded into the test environment where it was executed.

This relatively simple example illustrates several common situations within member companies. Modeling is sometimes the only way requirements are captured for applications. It is common for a few key technical staff members to understand the requirements that are not universally understood or documented. Model-based development fosters the development of interfaces that support automated test injection and test results analysis. It is also common for traditional testing, often manual, to be less than comprehensive. If testing and results analysis are done manually, developing and executing 110 test cases can take significant time and effort. This example illustrates that it is relatively straight- forward to capture the time card processing logic, and through the use of inlining mechanisms, generate all combinations of options in test cases that can be executed automatically. The example briefly illustrates the use of requirement management and requirement traceability linkages. As systems become more complex, requirement traceability is useful in assessing the completeness of the testing process, and it also speeds failure analysis.

11. FLIGHT GUIDANCE MODEL LOGIC

11.1 PROBLEM

Incomplete, ambiguous, or rapidly changing requirements can have a profound impact on the quality and cost of software development. In an effort to provide a more rigorous approach to flight-critical system development, Rockwell Collins used some formal approaches to develop the mode control logic of a Flight Guidance System (FGS) for a General Aviation class aircraft [Miller 1998]. However, they believed that the complexity of the system and models for real-world applications such as the FGS system would be too complex to analyze manually. They believed that tool support would be the only way to systematically assess such models.

This case study discusses the results of a Rockwell Collins study where they used an early version of TAF tools for model-based analysis and test automation to analyze the requirement model and generate tests for a new implementation of the FGS system.

11.2 APPROACH

Rockwell Collins used the Consortium Requirements Engineering Method (CoRE) [Faulk 1993] and SCR methods to specify the requirements for the mode logic of an FGS. An FGS compares the measured state of an aircraft (position, speed, and attitude) to the desired state and generates pitch and roll guidance commands to minimize the difference between the measured and desired state. The mode logic accepts commands from the flight crew and a variety of systems, such as the Flight Management System (FMS).

As reflected in Figure 48:

- FGS requirements were specified using variants of the SCR method.
- The first variant, referred to as CoRE, had no tool, but through a precise manual language and inspection technique, Rockwell Collins was able to identify 33 requirement defects.
- The second variant, based on a tool for SCR, revealed 27 additional defects through automated tool analysis.
- The third variant, based on T-VEC tools, revealed an additional six defects not previously identified.
- The fourth variant, based on Offutt method [Offutt 1999] revealed two defects.

- The fifth variant, based on a second generation of TAF tools, revealed an additional 25 defects for a total of 85 defects.

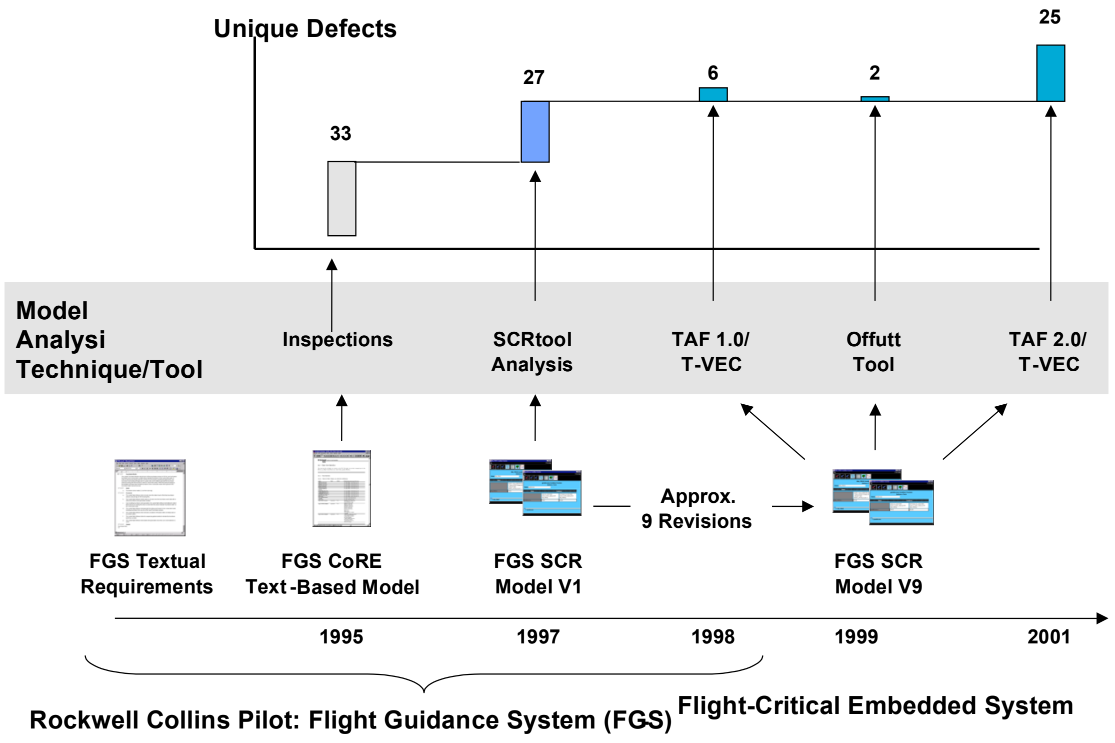


Figure 48. Model Evolution and Analysis

11.3 IMPLEMENTATION

The FGS model contains 78 SCR tables, including 47 condition tables, 14 mode transition tables, and 15 event tables. The FGS model was translated into 78 T-VEC subsystems, each corresponding to an SCR table. Translation and processing by the T-VEC tools produced 884 unique DCPs. 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. By default, T-VEC attempts to determine two test vectors for each DCP, one with low-bound values and another with high-bound values. Therefore, test generation should have produced 1,778 test vectors. However, because of latent errors remaining in the FGS SCR specification, only 1,700 test vectors were produced.

Table 3 summarizes the classes of defects identified. The first 2 rows indicate that 10 newly discovered model contradictions resulted in 25 model defects. The missing test vectors occurred because these contradictions produce unsatisfiable DCPs. Informally, from a test generation perspective, a specification is *satisfiable* if at least one test vector exists for every DCP. If a test vector is not produced for a DCP, it probably contains a contradiction (aka a requirement defect, a feature interaction problem). Each contradiction involved at least one event or condition table and at least one mode table. An additional 21 faults of various types in the implementation resulted in 95 test failures. The test cases also revealed six known bugs. For more details, see [Busser 2001].

Table 3. FGS Analysis Details

Defect Description	Defect Type	Unique Defects	Total Defects
Invalid event expression for related mode table	Model	5	7
Invalid constraint for dependency	Model	5	18
Mode transition implemented incorrectly	Code	2	3
Specification not implemented	Code	1	1
Variable referenced before set	Code	3	26
Incorrect implementation of mode logic	Code	1	4
Incorrect code (likely cut/paste error)	Code	1	1
Incorrect event implementation	Code	3	44
Hidden bug – coincidental correctness	Code	1	2
Unmodeled domain knowledge	Code	3	8
New Errors		25	114
Known bugs	Code	6	6
Total Errors Detected		31	120

11.4 KEY GUIDELINES

Over the 8 or so years of working with members, the TAF team has noticed that event specifications can be tricky to specify. A significant percentage of the modeling errors relate to the use of event specifications. It is important to consider the subtle implications of inherent states when using event specifications in models.

11.5 RESULTS

TAF and T-VEC identified defects missed by manual inspection and other tools and methods. Rockwell Collins' evaluation of the model-based analysis and test generation approach demonstrated to management that tools provide a more systematic and comprehensive approach to support cost-effective development of complex, safety-critical software. These situations are common as reflected by another member company (that prefers to remain anonymous) that had similar results. The pilot study, conducted by a member company, comparing formal Fagan inspections with TAF requirement verification, revealed that Fagan inspections uncovered 33 defects. In comparison, as shown in Figure 49, TAF uncovered all 33 of the Fagan inspection defects plus 56 more. Attempting to repeat the Fagan inspection did not improve its results. The improved defect detection of TAF prevented nearly two-thirds more defects from entering the rest of the development life cycle.

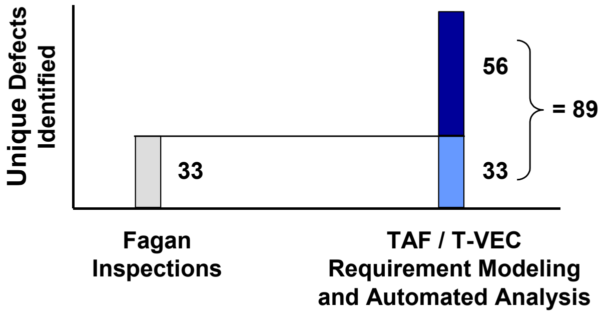


Figure 49. Fagan Inspections Versus TAF/T-VEC

More recent applications of T-VEC solutions by Rockwell Collins have demonstrated its total software development costs using a full model-based development environment, including autocode and autotest, could save up to 52% of development cost on safety-critical products. For more details, see [Busser 2000].

The results illustrate the importance of using tools to support the analysis of complex systems. As the analysis by Rockwell Collins points out, both applications and the associated requirements are too complex to analyze without the use of tools. This historical evolution of the original FGS model used inspections to remove model defects and the SCRtool model analysis capabilities to identify problems in individual tables. The first generation TAF/T-VEC tools and the Offutt tool were able to detect additional faults, including those related to multiple tables. The latest version of the TAF/T-VEC tools identified many model defects that primarily involved multiple tables. In addition, these tools helped uncover many additional faults in the implementation. Model defects and implementation faults similar to those discovered occur commonly in complex systems. As software-based systems continue to evolve, the capabilities demonstrated can provide greater assurance that these systems operate dependably.

12. DATABASE SECURITY

12.1 PROBLEM

The cost of developing and performing security functional testing is costly, and the increased demand for product variations is increasing the cost impacts on security evaluation laboratories. In addition, there is a need to better understand Security Functional Testing for various Security Functional Requirement (SFR) classes in ISO/IEC 15408 [ISO/IEC 1999]. NIST had an objective to develop a methodology and architecture for partially automating the security testing process for testing the security worthiness of a product or system.

Companies and government agencies use commercial products, and security is essential. Databases often are central to many enterprise systems, and the secure management of data is critical. In order to supply products to government organizations, vendors often must supply a Common Criteria Security Target (CST) for their specific product. This case study describes models and tests for the functional security properties derived from the Common Criteria of the Oracle8 Database Server, Release 8.0.5.

12.2 APPROACH

The CST defines the Target Of Evaluation (TOE) SFRs [Oracle 2000]. These requirements are mapped to TOE Security Functional Specifications that provide a more Oracle-based detailed functional specification that is traceable to the SFR (For details see Table 5 in the Oracle8 Security Target at http://www.commoncriteriaportal.org/public/files/epfiles/o80_st.pdf). The Oracle Corporation claims that the TOE Security Functionality (TSF) found in the Oracle Database Server meets the SFRs. The effort to demonstrate compliance with the security properties involves modeling the TSF, and then generating tests that are executed against the Oracle database for each requirement. These requirements are grouped into the following categories:

- Audit generation
- Identification and Authentication
- Security Management
- Session Management

The CST uses a naming convention for each requirement as specified in Table 5 of the Oracle8 Security Target at http://www.commoncriteriaportal.org/public/files/epfiles/o80_st.pdf. For example, the Grant Object Privilege specification is labeled F.APR.GOP, and the related specification for Revoke Object Privilege is labeled F.APR.ROP. The model for each functional specification is named using the label field; however, an underscore character is substituted for the period because the period character is not a valid character in the modeling tool. Therefore, F.APR.GOP is named F_APR_GOP in the new version of the specification.

An interface-driven approach is used to identify common tables and SQL commands, and then models are defined for the requirements in terms of those interfaces. This then supports common test driver mappings that can be extended and maintained as the product evolves. Table 4 summarizes the requirements for some of the specifications and identifies data dictionary items that are part of the interface associated with each specification.

Table 4. Mapping of Specifications to Interfaces

Requirement Area	Model Element	Data Dictionary Items	Requirement Summary
Object Access Control	F_APR_DER	DBA_ROLE_PRIVS SESSION_ROLES	Disable Roles
	F_APR_EDR	DBA_ROLE_PRIVS SESSION_ROLES	Enable roles
	F_APR_GOP	DBA_TAB_PRIVS	Grant object privileges
	F_APR_GRR	DBA_ROLE_PRIVS	Grant role privileges
	F_APR_GRSP	DBA_SYS_PRIVS	Grant system privileges
	F_APR_ROP	DBA_TAB_PRIVS	Revoke privileges
	F_DAC_OBID	ALL_OBJECTS-objects accessible to the user.	Every object uniquely identified, even if deleted
	F_DAC_OBA	USER_OBJECTS – objects owned by the user. USER_OBJECTS – objects accessible to the user.	TOE enforces data access control on objects based on object owner identify and granted privileges
	F_DAC_POL	DBA_TAB_PRIVS SESSION_PRIVS	Control of operation between subjects and objects based on rules: object owner, session privilege, system privilege, Database Administrator (DBA)
	F_DAC_SUA	USER_OBJECTS – objects owned by the user. ALL_OBJECTS – objects accessible to the user.	TOE enforces data access control on objects based on user session identify and session privileges
Identification and Authentication	F_IA_ATT	ALL_USERS – information about users of database. USER_USERS – information about current user. DBA_USERS – information about users of database. SESSION_PRIVS – privileges currently available to user. DBA_SYS_PRIVS – system privileges granted to users/roles. SESSION_ROLES – roles that are currently enabled to user. DBA_ROLE_PRIVS – roles granted to users and roles. USER_ROLE_PRIVS – roles granted to user. USER_RESOURCE_LIMITS – resource limits for current user.	Data dictionary contains a unique set of security attributes for each user, including their username, privileges, roles, and resource limits that can be displayed and modified by suitably privileged users using standard SQL commands.
	F_IA_CNF	Subsumed	Only a suitably authorized user can create a database user
	F_IA_IDE	Subsumed	TOE is able to establish the identity of the user
	F_IA_UID	ALL_USERS	Each database user is uniquely identified
Access Control	F_LIM_CNF	DBA_PROFILES – displays all profiles and their limits. DBA_USERS – information about all users of the database.	Alters the default Resource Profile for a database and creates and alters specific Resource Profile
	F_LIM_NSESS	V$LICENSE	TOE prevents a user from creating more than the maximum number of concurrent sessions specified for that user
	F_LIM_POL	DBA_PROFILES – displays all profiles and their limits. DBA_USERS – information about all users of the database	TOE enforces the limits specified by the resource profile or default profile.
	F_LIM_RSESS	DBA_PROFILES – displays all profiles and their limits. DBA_USERS – information about all users of the database.	TOE terminates operation or session if user attempts to perform an operation that exceeds the specified resource limits

12.3 IMPLEMENTATION

This section uses the Grant Object Privilege requirement specification as an example to describe the implementation of the functional security requirements. This Grant Object Privilege requirement should be understandable without in-depth knowledge of a database. Within a database object, the user can create such a database table. The user that creates the object is the owner and has privileges to access the table. In addition, the owner has the privilege to grant other users access to the table, including granting privilege to other users. The Grant Object Privilege requirement in the Oracle TOE states:

```
A normal user (the grantor) can grant an object privilege to another
user, role or PUBLIC (the grantee) only if:
  a) the grantor is the owner of the object; or
  b) the grantor has been granted the object privilege with the GRANT
OPTION.
```

Model variables are used to represent database tables, objects, privileges, and relationships. The SQL operations that are related directly to the granting of the object privileges include:

```
GRANT <privilege> ON <object> TO <user | role | PUBLIC> [WITH GRANT
OPTION]

Where <privilege> can be: ALTER, EXECUTE, INDEX, INSERT, READ,
REFERENCES, SELECT, UPDATE, ALL, and the GRANT OPTION is optional.

And, where <object> is a database schema object like a table, view,
sequence, procedure, function, package, or snapshots.

And, where <user> is a database user, <role> is a defined database
role, and <PUBLIC> represents all users.
```

However, there are some initial privileges and dependent SQL commands that are related to the GRANT SQL command. These involve the creation of a user, role, or session, as follows:

- When a user is created with the CREATE USER command, the user’s privilege is empty.
- To log on to Oracle, a user must have the CREATE SESSION system privilege. After creating a user, the user must be granted this privilege.

There are numerous other cases where additional constraints restrict grant privileges on various object types. These details are beyond the scope of this report and are not discussed.

The data dictionary table that is affected, or can be used to determine whether a particular GRANT operation is successful, is DBA_TAB_PRIVS, as reflected in Table 4. This data dictionary view lists all grant privilege details on objects in the database. It has attributes that indicate the GRANTEE (user to whom access was granted), object owner, name of the object, GRANTOR (user who performed the grant operation), privilege, and an indication of whether the privilege can be granted to another user.

As shown in Figure 50, the behavioral requirements are derived from the requirement text in the Oracle Security Target, like Grant Object Privilege, described in Section 12.3.1. The requirements are defined in terms of the model variables that represent the interface, which is defined in terms of the data dictionary and SQL commands. The interfaces are declared as model variables using the modeling tool. The mapping for the model variable defines how to affect that variable within the test execution environment. For example, a user or script must issue a GRANT SQL command to affect an object's privilege.

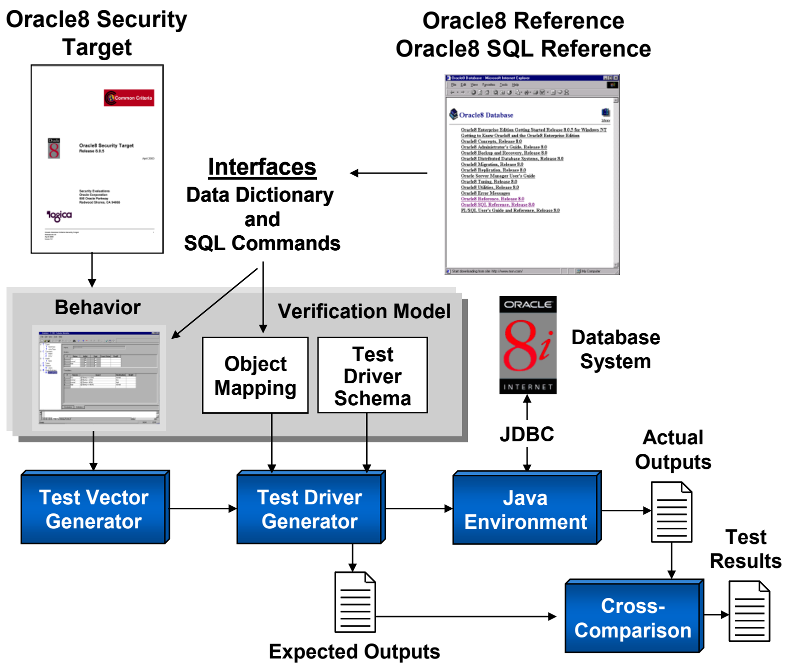


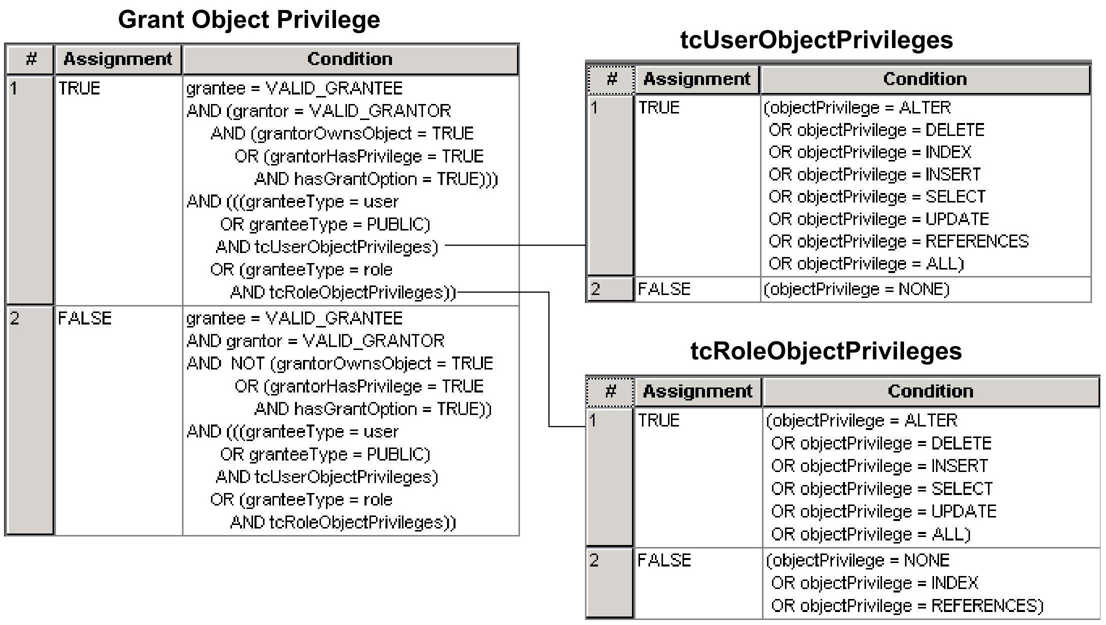
Figure 50. Detailed Process Flow

As shown in Figure 50, the model is input to the test vector generator, and the resulting test vectors are combined with the object mappings and test driver schema to produce a Java test driver. The executing test driver communicates with the Oracle database through a JDBC connection to carry out the tests. The test driver captures the actual outputs for each test during test execution and stores them for post processing. Then, a cross-comparison tool compares the expected outputs against the actual outputs and produces a test results log that indicates the pass/fail status for each test vector.

12.3.1 MODELING SECURITY PROPERTIES

Each security property is modeled as a Boolean object in a manner similar to Grant Object Privilege as shown in Figure 51. The conditions associated with the TRUE output, or the positive sense for the model, make up the valid set of conditions required for Grant Object Privilege. Each test case for the TRUE case should result in valid actions with respect to the security relationships established for that case. The FALSE cases are negative conditions, which establish a realistic database relationship, but the corresponding test attempts to execute invalid operations, from a security perspective, should be denied as an invalid security response. Some operations cause failures because the database responds with an error message when improper or unauthorized actions are requested. If any of the FALSE cases does not respond with some type of invalid operation, then the security property has been violated. This general approach is used to model each security requirement to ensure that proper security exists for authorized actions, while unauthorized actions are not permitted.

Row 1 of the model for Grant Object Privilege, shown in Figure 51, with the assignment TRUE, describes the conditions when the Grant Object Privilege should be permitted. For example, when the grantee and the grantor are valid database users, then an object privilege should be granted if the grantor owns the object or has been granted object privileges with the GRANT OPTION. Also, the model defines additional conditions where the grantee (reflected by granteeType) can be a user, PUBLIC, or role. The term variable tcUserObjectPrivileges references another condition table that enumerates the set of objectPrivileges (e.g., ALTER, DELETE, INDEX, and INSERT) that are valid and should be tested. If the granteeType is a role, then the term tcRoleObjectPrivileges defines a subset of the valid ObjectPrivileges that apply to roles.



Grant Object Privilege

#	Assignment	Condition
1	TRUE	grantee = VALID_GRANTEE AND (grantor = VALID_GRANTOR AND (grantorOwnsObject = TRUE OR (grantorHasPrivilege = TRUE AND hasGrantOption = TRUE))) AND (((granteeType = user OR granteeType = PUBLIC) AND tcUserObjectPrivileges) OR (granteeType = role AND tcRoleObjectPrivileges))
2	FALSE	grantee = VALID_GRANTEE AND grantor = VALID_GRANTOR AND NOT (grantorOwnsObject = TRUE OR (grantorHasPrivilege = TRUE AND hasGrantOption = TRUE)) AND (((granteeType = user OR granteeType = PUBLIC) AND tcUserObjectPrivileges) OR (granteeType = role AND tcRoleObjectPrivileges))

tcUserObjectPrivileges

#	Assignment	Condition
1	TRUE	(objectPrivilege = ALTER OR objectPrivilege = DELETE OR objectPrivilege = INDEX OR objectPrivilege = INSERT OR objectPrivilege = SELECT OR objectPrivilege = UPDATE OR objectPrivilege = REFERENCES OR objectPrivilege = ALL)
2	FALSE	(objectPrivilege = NONE)

tcRoleObjectPrivileges

#	Assignment	Condition
1	TRUE	(objectPrivilege = ALTER OR objectPrivilege = DELETE OR objectPrivilege = INSERT OR objectPrivilege = SELECT OR objectPrivilege = UPDATE OR objectPrivilege = ALL)
2	FALSE	(objectPrivilege = NONE OR objectPrivilege = INDEX OR objectPrivilege = REFERENCES)

Figure 51. Example Model for Grant Object Privilege

12.3.2 TEST VECTORS

Modeling and test vector generation are typically performed iteratively as the model is developed. The TTM modeling tool provides a number of checks on the model to ensure that individual tables are consistent and complete. The TAF model translator and T-VEC tools perform additional checks that identify cross-table inconsistencies and contradictions. These model analysis capabilities support refining the model by identifying and correcting model defects.

The Grant Object Privilege requirement includes 88 requirement threads to cover the combinations of object privileges and roles. The test vector generator attempts to determine two test vectors for each requirement thread based on a test selection strategy derived from the concept of domain testing theory. Table 5 shows a tabular representation of the 176 test vectors that have been compressed into 132 test cases. If one ore more test vectors have the same input and output values, they are compressed into a single test case. The test vectors include seven input variables. The test values shown in Table 5 reflect how the test generator systematically selects low-bound and high-bound test points at the domain boundaries. The input values, ranges, and constraints (e.g., relational operators) of the specification define the domain boundaries. For example, vector 1, grantor = 1, grantee = 2, is based on low-bound values of the data type range of userIDType. The other inputs such as granteeType toggle between the different values of user, PUBLIC, and role, as do the other inputs such as objectPrivilege.

Table 5. Test Vectors for Grant Object Privilege

Test #	Vector #'s	_output	grantee	granteeType	grantor	grantorHasPrivilege	grantorOwnsObject	hasGrantOption	objectPrivilege
1	4.7	**TRUE=1**	2	user=0	1	TRUE=1	TRUE=1	TRUE=1	ALTER=0
2	2	**TRUE=1**	2	user=0	1	FALSE=0	TRUE=1	FALSE=0	ALTER=0
3	3.9	**TRUE=1**	2	PUBLIC=2	1	TRUE=1	TRUE=1	TRUE=1	ALTER=0
4	4	**TRUE=1**	2	PUBLIC=2	1	FALSE=0	TRUE=1	FALSE=0	ALTER=0
5	5.11	**TRUE=1**	2	role=1	1	TRUE=1	TRUE=1	TRUE=1	ALTER=0
6	6	**TRUE=1**	2	role=1	1	FALSE=0	TRUE=1	FALSE=0	ALTER=0
7	8	**TRUE=1**	2	user=0	1	TRUE=1	FALSE=0	TRUE=1	ALTER=0
8	10	**TRUE=1**	2	PUBLIC=2	1	TRUE=1	FALSE=0	TRUE=1	ALTER=0
9	12	**TRUE=1**	2	role=1	1	TRUE=1	FALSE=0	TRUE=1	ALTER=0
10	13.19	**TRUE=1**	2	user=0	1	TRUE=1	TRUE=1	TRUE=1	DELETE=1
126	167	**FALSE=0**	2	PUBLIC=2	1	TRUE=1	FALSE=0	FALSE=0	INDEX=2
127	169	**FALSE=0**	2	user=0	1	FALSE=0	FALSE=0	TRUE=1	REFERENCES=4
128	170.174	**FALSE=0**	2	user=0	1	FALSE=0	FALSE=0	FALSE=0	REFERENCES=4
129	171	**FALSE=0**	2	PUBLIC=2	1	FALSE=0	FALSE=0	TRUE=0	REFERENCES=4
130	172.176	**FALSE=0**	2	PUBLIC=2	1	FALSE=0	FALSE=0	FALSE=0	REFERENCES=4
131	173	**FALSE=0**	2	user=0	1	TRUE=1	FALSE=0	FALSE=0	REFERENCES=4
132	175	**FALSE=0**	2	PUBLIC=2	1	TRUE=1	FALSE=0	FALSE=0	REFERENCES=4

The complete model resulted in 382 test vectors for 234 DCPs.

12.4 KEY GUIDELINES

This section summarizes several key guidelines derived from this case study, including:

- Create data dynamically at test time
- Use interface-driven analysis and modeling
- Model positive and negative cases
- Use test vector compression to reduce redundant test cases

12.4.1 Creating a Test-Time Database

One of the challenges in testing large applications, especially those with databases, involves the creation of test data. Often "gold" databases are created, and test cases must be defined in terms of the information stored within those databases. Creating the "gold" data can be costly, but to create test cases for the data within the database, those values must be analyzed in order to create relevant test cases. This adds another level of complexity to the task of security testing. If the "gold" data changes, then existing test cases often become invalid, and new tests must be constructed.

One key advantage of model-based testing is that the test case data is created dynamically at test time. The test driver for this application dynamically creates and deletes database information in the form of users, roles, database tables, and values. This allows automated test execution without manual assistance. There is upfront effort that must be performed once, similar to the creation of a gold database. However, this initial effort eliminates the need to maintain the gold database, and the user does not need to analyze the gold database before creating test cases. The models are constructed in a way that is independent of any specific populated database.

For any Oracle database, some specific database conditions must be established prior to the execution of the tests. For example, a database administrator must install the database, and the Oracle database test execution requires the TEMPORARY tablespace to be available during execution. Once this is established, the common sequences of tests are produced from model. The following simplified example, taken from a log file for the grant role command, illustrates the sequence of commands to log in as a user and initialize the data by dropping roles, creating tables, and inserting data. The description follows the example:

- Logon User -> SYSTEM
- Create Users
```
1. Executed SQL-> CREATE USER "USER1" IDENTIFIED BY "USER1" DEFAULT
   TABLESPACE "TABLESPACE" QUOTA UNLIMITED ON "TABLESPACE" TEMPORARY
   TABLESPACE "TABLESPACE" PROFILE DEFAULT ACCOUNT UNLOCK
2. Executed SQL-> GRANT CONNECT TO USER1
3. Executed SQL-> CREATE USER "USER2" IDENTIFIED BY "USER2" DEFAULT
   TABLESPACE "TABLESPACE" QUOTA UNLIMITED ON "TABLESPACE" TEMPORARY
   TABLESPACE "TABLESPACE" PROFILE DEFAULT ACCOUNT UNLOCK
4. Executed SQL-> GRANT CONNECT TO USER2
5. Executed SQL-> CREATE USER "USER3" IDENTIFIED BY "USER3" DEFAULT
   TABLESPACE "TABLESPACE" QUOTA UNLIMITED ON "TABLESPACE" TEMPORARY
   TABLESPACE "TABLESPACE" PROFILE DEFAULT ACCOUNT UNLOCK
6. Executed SQL-> GRANT CONNECT TO USER3
7. Executed SQL-> CREATE USER "USER4" IDENTIFIED BY "USER4" DEFAULT
   TABLESPACE "TABLESPACE" QUOTA UNLIMITED ON "TABLESPACE" TEMPORARY
   TABLESPACE "TABLESPACE" PROFILE DEFAULT ACCOUNT UNLOCK
8. Executed SQL-> GRANT CONNECT TO USER4
```
- Create Role
```
9.  Executed SQL-> CREATE ROLE ROLETEST
10. Executed SQL-> GRANT ROLETEST TO USER1
11. Executed SQL-> SELECT GRANTED_ROLE FROM SYS.DBA_ROLE_PRIVS where GRANTEE
    = 'USER1' and granted_role = 'ROLETEST'
```
- test:1 Results:TRUE
```
12. Executed SQL-> SELECT USERNAME from sys.ALL_USERS where USERNAME like
    'USER%'
```

```
13. Executed SQL-> DROP USER USER1 CASCADE
14. Executed SQL-> DROP USER USER2 CASCADE
15. Executed SQL-> DROP USER USER3 CASCADE
16. Executed SQL-> DROP USER USER4 CASCADE
17. Executed SQL-> SELECT unique profile FROM SYS.DBA_PROFILES where profile
    like 'PROFILE%'
18. Executed SQL-> SELECT role FROM SYS.DBA_ROLES where role like 'ROLE%'
19. Executed SQL-> DROP ROLE ROLETEST
20. Executed SQL-> SELECT OWNER, TABLE_NAME from sys.ALL_TABLES where
    TABLE_NAME like 'TABLE%'
```

- On lines 1 through 8, SYSTEM user creates users (1 to 4) and grants connection privileges.
- On lines 9 and 10, SYSTEM creates role and grants the role to user1.
- ON line 11, SYSTEM extracts database role privileges (DBA_ROLE_PRIVS) from the system data dictionary.
- On lines 12 through 20 SYSTEM performs cleanup operations.

The test sequence just shown was extracted from a log file that is produced as part of the test driver infrastructure. The test driver is based on the JDBC API, using Java that makes SQL calls to the database. The test driver generation support capabilities are provided by a Java infrastructure to do the following:

- Retrieve global test configuration settings that can be configured to direct the test driver mechanisms to use user-specified options such as log directory, output file directory, system user, and password

- Retrieve test vector parameters during test execution

- Log the test operation (shown in the previous example)

- Create a test output file

- Establish an Oracle database connection and SQL execution through JDBC

- Specify an interface to which each test must conform

- Provide global constants

- Provide a framework for test execution

The combination of dynamic test data creation and test infrastructure provides a robust environment to fully automate an ever-expanding set of tests as the model evolves.

12.4.2 Interface-Driven Modeling

Interface-driven analysis and the associated modeling are important to understand potentially common interface objects or variables as well as dependencies in the interface operations that are used to put a system into a test state. As illustrated in Section 12.4.1, the GRANT commands depend on other commands such as CREATE, INSERT, and SELECT. Table 6 provides a summary resulting from interface analysis for several requirements. Each row provides a brief summary of a requirement, the related data dictionary views, associated SQL commands that are primarily used to affect the operation, and related commands that are referred to as dependent commands.

For example, the Grant Role Privilege command, like the Grant Object Privilege command, describes the requirements for granting and revoking role privileges. The primary data dictionary table from which the results of the granted role privilege can be retrieved is the DBA_ROLE_PRIVS (database administrator role privileges). The SQL commands that are used to grant/revoke privileges are GRANT and REVOKE, and the related SQL commands include CREATE, INSERT, SELECT, and others. The operations and test driver commands required to support Grant Role Privilege overlap Grant Object Privilege. More importantly, much of the functionality for other requirements, such as DISABLE and ENABLE roles, subsume many of the tested requirements developed for GRANT and REVOKE roles.

Table 6. Detailed Security Specification Analysis

Requirement Summary	Data Dictionary Items	SQL Command	Dependant Command
Disable roles	DBA_ROLE_PRIVS SESSION_ROLES	SET ROLE	GRANT, ALTER,
Enable roles	DBA_ROLE_PRIVS SESSION_ROLES	SET ROLE	GRANT
Grant object privileges	DBA_TAB_PRIVS	GRANT	CREATE, INSERT, SELECT
Grant/revoke role privileges	DBA_ROLE_PRIVS	GRANT/ REVOKE	CREATE, INSERT, SELECT
Grant system privileges	DBA_SYS_PRIVS	GRANT	CREATE, INSERT, SELECT
Revoke privileges	DBA_TAB_PRIVS	REVOKE	GRANT
Every object uniquely identified, even if deleted	ALL_OBJECTS		CREATE, INSERT, SELECT, DELETE

Understanding the dependencies allows actual interfaces to be associated with modeled variables. These modeled variables are then related to common object mappings. Once an object mapping is created for an interface such as the CREATE SQL command, it can be reused, which saves time and effort and minimizes the cost.

12.4.3 Modeling Positive and Negative Conditions

The typical, and manual test case developed usually results in positive test cases that are similar to those represented by the TRUE assignments in Figure 51. The modeling process helps make the negative case more obvious. The FALSE cases, shown in Figure 51, are negative conditions, which establish realistic database relationships, but the corresponding test attempts to execute invalid operations, from a security perspective, and should be denied as an invalid security response. These are critical test cases from a security perspective, but this type of negative case is the same type as the model that found the MPL bug as discussed in Section 4.4.

12.5 RESULTS

This case study summarizes the general process for modeling a security requirement and the use of generated tests derived from the model to support automated testing of the security properties of an Oracle database. Modeling functional security properties is conceptually similar to modeling any other type of functional requirement. Mapping the generated tests to the interfaces of the database allows users to systematically exercise the various database configurations for each of the security properties.

The details of the models, test vectors, and test drivers are beyond the scope of this report. In addition, to understand the test driver support requires some understanding of Java, SQL, and operational details of an Oracle database. Additional details, including the security requirement models, test vectors, object mappings, test driver schema, test drivers, and instructions for installing and executing the test drivers against an Oracle database, are available for download from the TAF Reports website [http://www.software.org/pub/taf/Reports.html].

This case study provides evidence that model-based testing of functional security requirements and automated test executions are feasible and cost-effective. Once a model is created from a specification, it can evolve with the specification and be used to systematically reverify a system after every new release with significantly reduced costs.

In addition, the Java-based test infrastructure that was developed for this application was ported forward and used in the smart card application described in Section 13.

Security functional testing is a costly activity, and the continuous stream of product variations and releases add to the cost of reverification. As pointed out in other case studies, regression testing through model-based testing can reduce the cost and time by 50% or more over manual regression testing. In addition, organizations that are faced with manual regression testing, often choose not to be complete in the retesting process, but for mission-critical systems that rely on security, choosing to neglect some tests can be a risk.

This page intentionally left blank

13. SMART CARD INTEROPERABILITY

13.1 PROBLEM

Smart cards are being used to provide security for many types of applications. With an estimated market of 3.3 billion in 2005, their usefulness is based on their intrinsic portability and security. Smart cards have the ability to provide far more efficient, secure, and hard-to forge credentials than are currently used in the U.S. Smart cards can provide efficient, secure, and portable storage for medical, financial, and other personal information. However, there are many different manufactures of smart cards, and most cards are not interoperable.

A typical configuration for a smart card system consists of a host computer with one or more smart card readers attached to hardware communications ports. Smart cards can be inserted into the readers, and software running on the host computer communicates with these cards using a protocol defined by ISO 7816-4 [ISO 1995b] and ISO 7816-8 [ISO 1995a]. The ISO standard smart card communications protocol defines Application Protocol Data Units (APDUs) that are exchanged between smart cards and host computers.

Client applications traditionally have been designed to communicate with ISO smart cards using the APDU protocol through low-level software drivers that provide an APDU transport mechanism between the client application and a smart card. Smart card families can implement the APDU protocol in a variety of ways, so client applications must have intimate knowledge of the APDU set of the smart card with which they are communicating. Generally, this is accomplished by programming a client application to work with a specific card because it would not be practical to design a client application to accommodate the different APDU sets of a large number of smart card families.

The tight coupling between client applications and smart card APDU sets has several drawbacks. Application programmers must be thoroughly familiar with smart card technology and the complex APDU protocol. If the card application is hard-coded and becomes commercially unavailable, programmers must redesign the application to use different cards. Customers also have less freedom to select different smart card products because their applications will work only with one or a small number of similar cards. This GSC-IS provides solutions to a number of the interoperability challenges associated with smart card technology.

NIST initiated the Smart Card Interoperability Program to provide standards and tests to accelerate the use of this technology. This case study describes the approach and results of a model-based development and test generation effort that created models for the GSC-IS. It also describes the generated tests and test infrastructure that have been used to test the conformance of the Java language binding of the Basic Services Interface of the GSC-IS.

13.2 APPROACH

13.2.1 ARCHITECTURAL OVERVIEW

Figure 52 represents the conceptual architecture and behavior covered in the GSC-IS. The two key elements that require models, tests, and supporting test infrastructure are defined as the Basic Service Interface (BSI) and Virtual Card Edge Interface (VCEI), also referred to as the Service Provider Software (SPS).

The VCEI includes two sets of APDU commands: (1) an GSC-IS APDU set for use in conformant file system cards, and (2) a set of virtual machine (VM) APDUs for use in VM cards. The card edge also consists of the Card Capability Container (CCC), which is a file located on each conformant smart card, and the GSC-IS APDU mapping mechanism.

The GSC-IS ISO-conformant APDU set can be implemented directly by conformant cards (such as in a conformant file system card or as a VM card applet). It is expected that some file system smart cards may use native APDU instruction sets that differ from the GSC-IS APDU set. In those cases, an SPS must modify the ADPU set so that it conforms to the smart card’s native APDU set.

This section discusses the models used to specify and test the BSI interface methods only (i.e., not the VCEI functions). In addition, it discusses the approach used to construct test middleware for executing the functions against a simulator of the smart card.

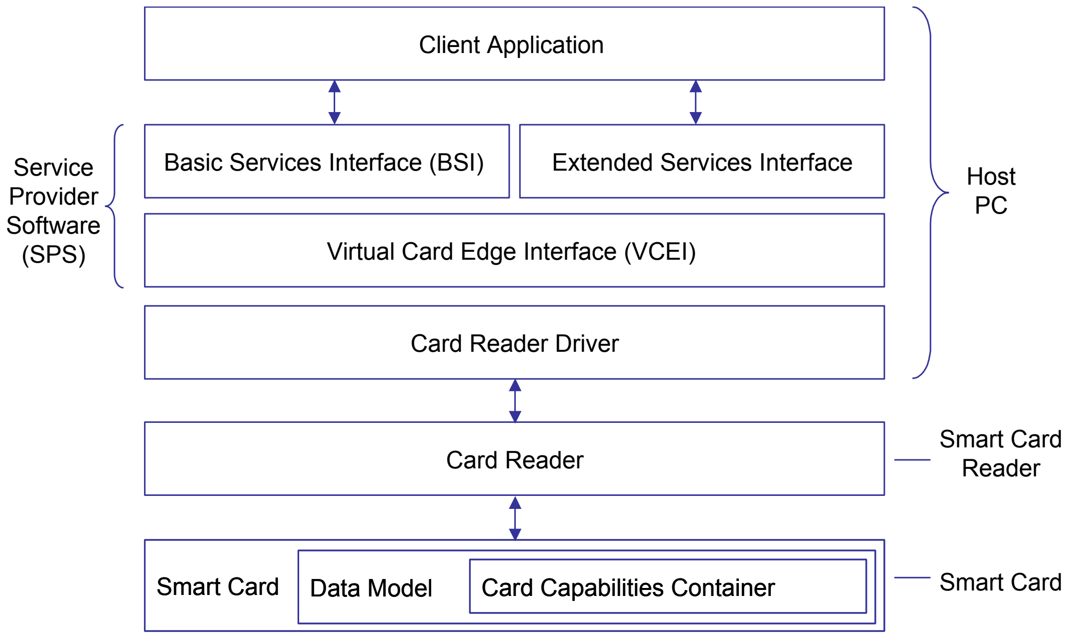

Client Application
Service Provider Software (SPS)
Basic Services Interface (BSI)
Extended Services Interface
Host PC
Virtual Card Edge Interface (VCEI)
Card Reader Driver
Card Reader
Smart Card Reader
Smart Card
Data Model
Card Capabilities Container
Smart Card

Figure 52. GSC-IS Architectural Model

13.2.2 ELEMENTS OF SYSTEM AND TEST INFRASTRUCTURE

The following elements, shown in Figure 53, were constructed to support the modeling and test analysis effort. A model of the GSC-IS, derived to characterize the functionality of the BSI and VCEI, was created using the TTM, based on the SCR method. The model was translated, and test vectors and test drivers were generated using the T-VEC test generation system for both the Java and C-language bindings. The Smart Card Conformance Tester is used to define configuration information and initiate executing generated tests. The Smart Card Model Maker creates data model information for a Smart Card Simulator. Information such as security information must be correlated between the configuration file and the defined smart card model. The Smart Card Conformance Tester executes the generated test drivers causing interaction with an implementation of the GSC-IS.

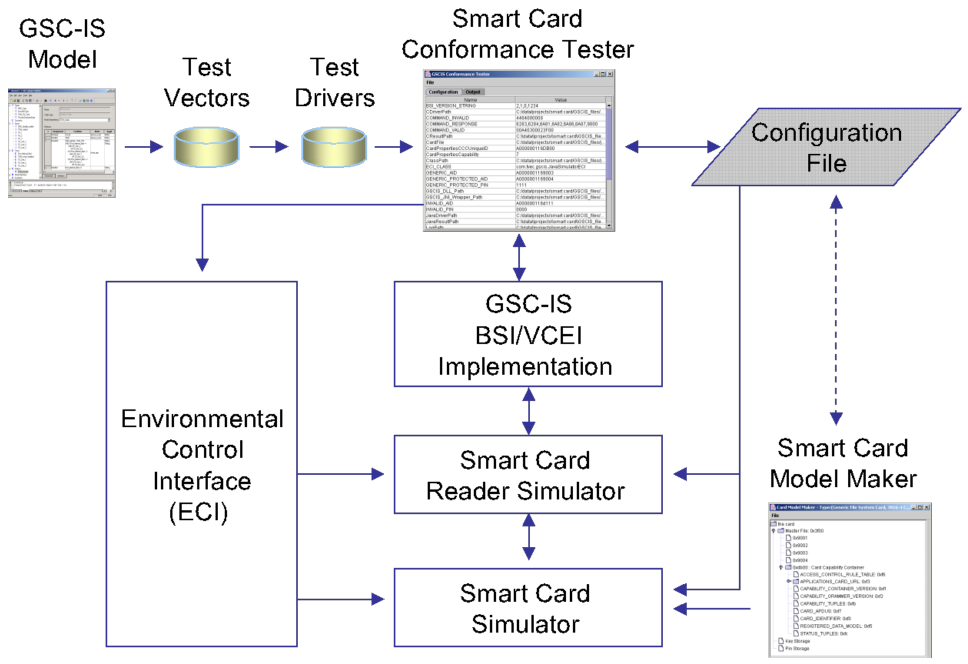

GSC-IS Model
Test Vectors
Test Drivers
Smart Card Conformance Tester
Configuration File
Environmental Control Interface (ECI)
GSC-IS BSI/VCEI Implementation
Smart Card Reader Simulator
Smart Card Simulator
Smart Card Model Maker

Figure 53. Elements Created to Support Task Development

The simulator implementation of the Environment Control Interface (ECI) interacts with the Smart Card Reader Simulator and Smart Card Simulator to support functions such as inserting and removing cards and attaching card readers. During test execution, the default implementation of the ECI prompts for the following:

- Card reader attachment
- Card insertion
- Card removal

13.3 IMPLEMENTATION

As shown in Figure 54, the testing of the GSC-IS involves:

- Building an SCR model specification for the GSC-IS using TTM
- Translating the TTM model into T-VEC specifications
- Generating the test vectors from the T-VEC specifications

- Building a test driver middleware to execute the test vectors against an implementation of the GSC-IS
- Executing the test driver
- Evaluating test results and producing a test results report

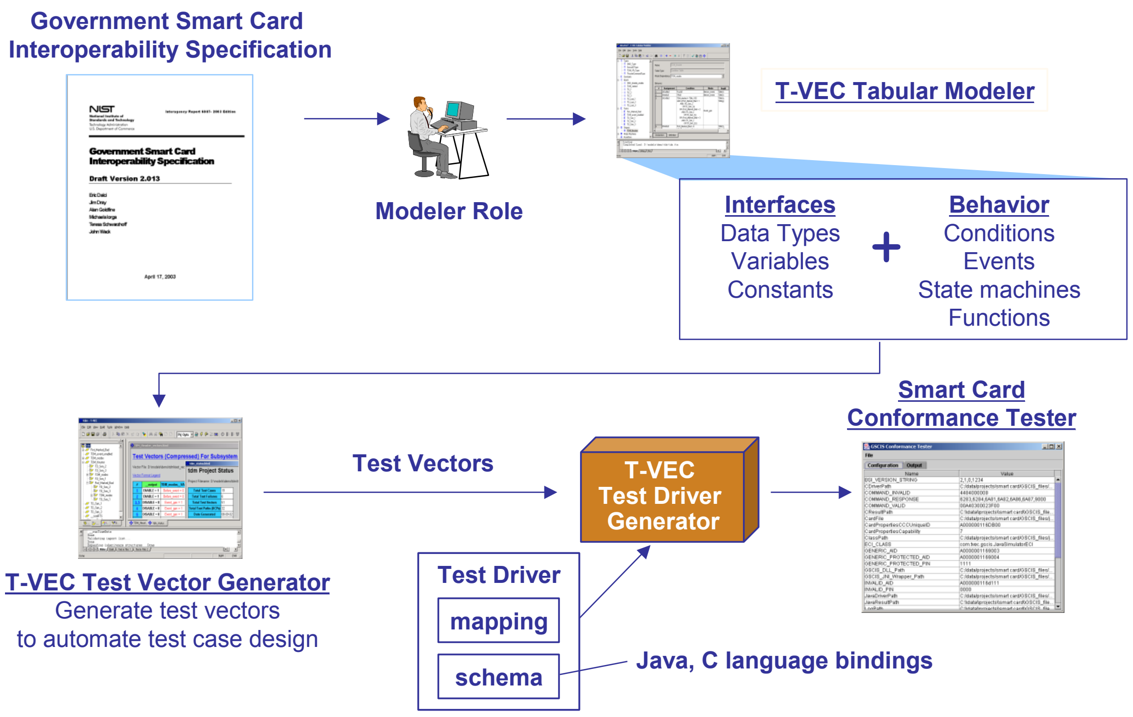

Government Smart Card Interoperability Specification
Modeler Role
T-VEC Tabular Modeler
Interfaces
Data Types
Variables
Constants
Behavior
Conditions
Events
State machines
Functions
Smart Card Conformance Tester
Test Vectors
T-VEC Test Driver Generator
T-VEC Test Vector Generator
Generate test vectors to automate test case design
Test Driver
mapping
schema
Java, C language bindings

Figure 54. GSC-IS Smart Card Modeling Process Overview

The recommended process involves analyzing the interfaces (inputs and outputs), and for each output, working backward from the required outputs to identify the conditions, events, and modes that define how each output is produced. For the GSC-IS, all operations are handled through the BSI method calls. Each method has a set of return codes that are output to the client application. Some operations cause data resident on the smart card to be created, deleted, or updated. Therefore, the categories of model outputs include the following:

- ***BSI Return Codes***. There are 23 condition tables, one for each of the BSI methods.
- ***No Card Services Outcomes***. There are 12 condition tables. The BSI_NO_CARDSERVICE error codes are returned when the inserted card does not support the services to complete (or possibly never allow) a successful GSC-IS BSI method call. Methods such as gscBsiPkiCompute, gscSkiInternalAuthenticate, and gscBsiGcDataCreate can return this error. This can occur if a card does not support a VCEI call.
- ***No Service Provider Software Outcomes***. Smart cards are not required to support transactions. There are two condition tables that deal with the GSC-IS transaction methods. The BSI_NO_SPSSERVICE return codes are returned by the

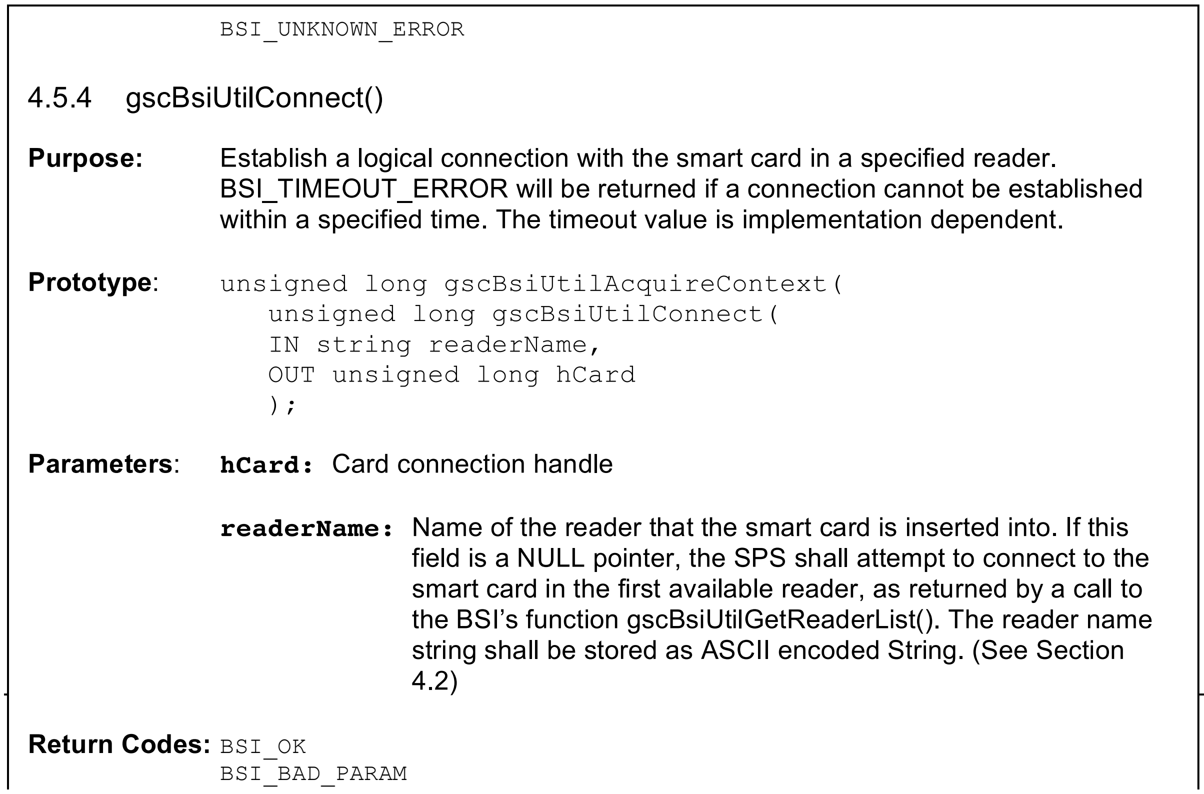
gscBsiUtilBeginTransaction and gscBsiUtilEndTransaction calls when the SPS does not provide transaction handling.

- ***Validation of Side-Effects to BSI Method Sequence Calls***. There are 14 condition tables that deal with verification of data that is operated on within the smart card or returned to the client application.
- ***Bad Card Outcomes***. There is one condition table that deals with a bad card outcome. These test drivers execute the SPS against a card that does not support the GSC-IS.

The GSC-IS specification presents each BSI method specification in a form similar to the example shown in the textbox insert. To perform operations on a smart card, an application must make a sequence of BSI method calls as represented in Figure 55. For example, starting from an initial state, an application must use the following sequence of calls to transition through the various states before it can operate on data stored within a smart card:

- gcsBsiUtilConnect
- gcsBsiGetContainerProperties (optionally a card also may be involved in multiple transactions)
- gcsBsiUtilAcquireContext
- gcsBsiGcReadTagList
- gcsBsiGcReadValue or gcsBsiGcUpdateValue
- gcsBsiGcDataCreate or gcsBsiGcDataDelete

At any state, it is also possible for the card to be removed from a card reader, resulting in a disconnect.



```
BSI_UNKNOWN_ERROR
```

4.5.4 gscBsiUtilConnect()

Purpose: Establish a logical connection with the smart card in a specified reader. BSI_TIMEOUT_ERROR will be returned if a connection cannot be established within a specified time. The timeout value is implementation dependent.

Prototype:
```
unsigned long gscBsiUtilAcquireContext(
   unsigned long gscBsiUtilConnect(
   IN string readerName,
   OUT unsigned long hCard
   );
```

Parameters: **`hCard:`** Card connection handle

`readerName:` Name of the reader that the smart card is inserted into. If this field is a NULL pointer, the SPS shall attempt to connect to the smart card in the first available reader, as returned by a call to the BSI's function gcsBsiUtilGetReaderList(). The reader name string shall be stored as ASCII encoded String. (See Section 4.2)

Return Codes:
```
BSI_OK
BSI_BAD_PARAM
```

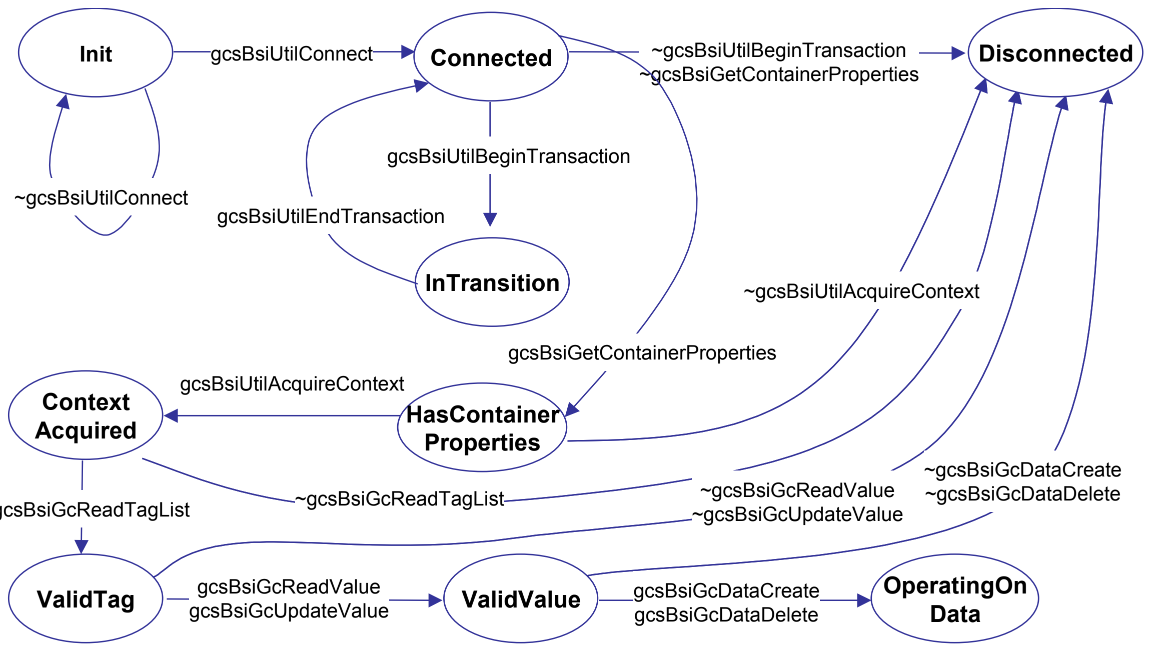
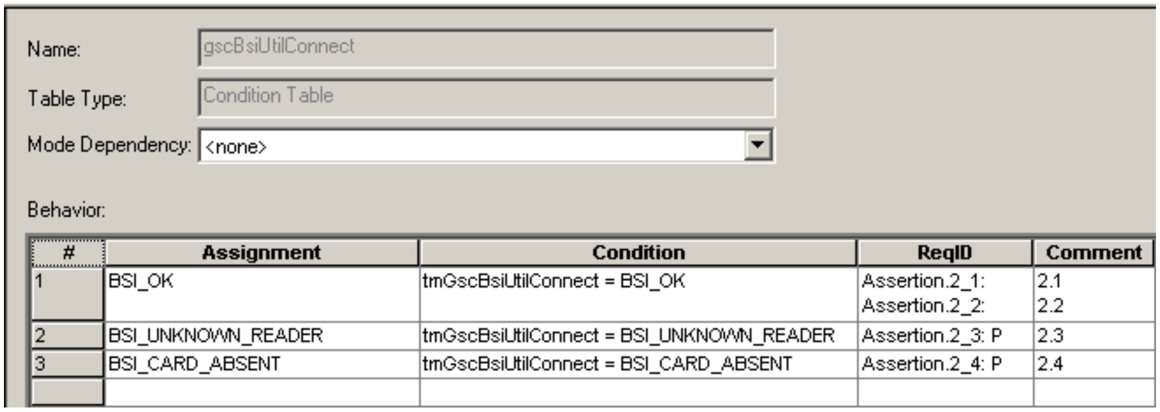


Figure 55. Partial State Representation of BSI

Although Figure 55 looks like a state transition diagram, it is not possible to put a BSI implementation into a particular state. It is necessary to make an appropriate sequence of BSI method calls to transition the implementation to various states. However, the conditions required of the inputs to transition to the states for the various sequences are common. Therefore, these common conditions are modeled as SCR terms that can be reused by other condition table outputs.

A condition table is created and named the same as the method name for each GSC-IS output. Similarly, the term related to the BSI method associated with the output is prefixed with a “tm” followed by the name of the BSI method. For example, the model shown in Figure 56 reference the term table table tmGscBsiUtilConnect for each of the three testable return codes. Types are defined for each input/output that is not represented as a true/false value. tBsiReturnCodes is an example of a type and represents an enumeration of all the GSC-IS return codes (e.g., BSI_OK, BSI_UNKNOWN_READER).



Name: gscBsiUtilConnect

Table Type: Condition Table

Mode Dependency: <none>

Behavior:

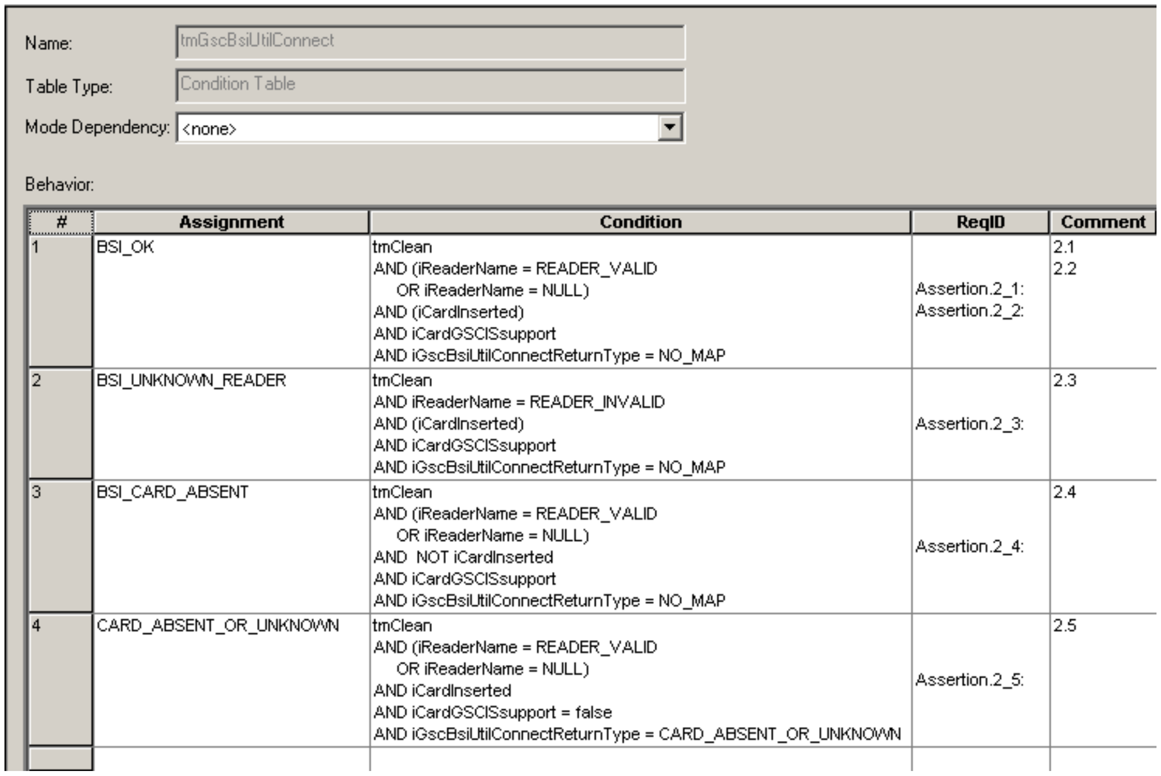
#	Assignment	Condition	ReqID	Comment
1	BSI_OK	tmGscBsiUtilConnect = BSI_OK	Assertion.2_1: Assertion.2_2:	2.1 2.2
2	BSI_UNKNOWN_READER	tmGscBsiUtilConnect = BSI_UNKNOWN_READER	Assertion.2_3: P	2.3
3	BSI_CARD_ABSENT	tmGscBsiUtilConnect = BSI_CARD_ABSENT	Assertion.2_4: P	2.4

Figure 56. Condition Table for gscBsiUtilConnect

The term table tmGscBsiUtilConnect shown in Figure 57 characterizes the inputs that are required for each of the return code values. For example, the return code value should be:

```
BSI_OK when

  tmClean
   AND (iReaderName = READER_VALID
        OR iReaderName = NULL)
   AND (iCardInserted)
   AND iCardGSCISsupport
   AND iGscBsiUtilConnectReturnType = NO_MAP
```

The term tmClean maps to test driver control for initializing and putting the client (i.e., the conformance tester in this case) into a valid starting state. In addition, this expression of the term models the situation where the reader name is valid or null, and a card supporting the GSC-IS is inserted into the reader. If there is a valid card reader and card ready, and there is not a time out or an unknown error, the method should return BSI_OK and then disconnect. In addition, various error conditions often are established to represent an input to the subsystem. This is done because part of the model must represent elements of the test environment, such as the availability of a card, or a card in or not in the reader. All external inputs needed to set up a test also are included in the test conditions for a test. This allows each test vector to define a mapping to the test environment in which it should be executed. In testing a GSC-IS implementation, these external inputs are used to connect readers, initialize cards, and insert cards.



Name: tmGscBsiUtilConnect

Table Type: Condition Table

Mode Dependency: <none>

Behavior:

#	Assignment	Condition	ReqID	Comment
1	BSI_OK	tmClean AND (iReaderName = READER_VALID OR iReaderName = NULL) AND (iCardInserted) AND iCardGSCISsupport AND iGscBsiUtilConnectReturnType = NO_MAP	Assertion.2_1: Assertion.2_2:	2.1 2.2
2	BSI_UNKNOWN_READER	tmClean AND iReaderName = READER_INVALID AND (iCardInserted) AND iCardGSCISsupport AND iGscBsiUtilConnectReturnType = NO_MAP	Assertion.2_3:	2.3
3	BSI_CARD_ABSENT	tmClean AND (iReaderName = READER_VALID OR iReaderName = NULL) AND NOT iCardInserted AND iCardGSCISsupport AND iGscBsiUtilConnectReturnType = NO_MAP	Assertion.2_4:	2.4
4	CARD_ABSENT_OR_UNKNOWN	tmClean AND (iReaderName = READER_VALID OR iReaderName = NULL) AND iCardInserted AND iCardGSCISsupport = false AND iGscBsiUtilConnectReturnType = CARD_ABSENT_OR_UNKNOWN	Assertion.2_5:	2.5

Figure 57. Condition Table for tmGscBsiUtilConnect

Another example that better illustrates the reuse of the terms is gscBsiPkiGetCertificate. This method has return code outputs, a No Card Service output, and returns a buffer containing the Public Key Infrastructure (PKI) certificate. Figure 58 provides a partial representation of the dependency structure of the modeled requirements for the outputs and some of terms. For example, there are three condition tables related to the different outputs: gscBsiPkiGetCertificate, noCardServicePkiGetCertificate, and validationPkiGetCertificate. These three tables depend on a common term tmGscBsiPkiGetCertificate, and this term depends on tmGscBsiUtilAcquireContext, which depends on other terms that descend from tmGscBsiGetCryptoProperties and tmGscBsiGcGetContainerProperties. shows the condition table for gscBsiPkiGetCertificate. The more interesting specification is defined for the term tmGscBsiPkiGetCertificate shown in the text box.

4.5.4 gscBsiPkiGetCertificate()

Purpose: Reads the certificate from the smart card

Prototype:

```
unsigned long gscBsiPkiGetCertificate(
    IN unsigned long hCard,
    IN string AID,
    INOUT sequence<byte> Certificate
    );
```

Parameters: **`hCard:`** Card connection handle

`AID:` PKI provider module Application Identifiers (AID) value. The parameter shall be in ASCII hexadecimal format.

`certificate:` Buffer containing the certificate

Return Codes:

```
BSI_OK
BSI_BAD_HANDLE
BSI_BAD_AID
BSI_CARD_REMOVED
BSI_SC_LOCKED
BSI_NO_CARDSERVICE
BSI_ACCESS_DENIED
BSI_IO_ERROR
BSI_INSUFFICIENT_BUFFER
```

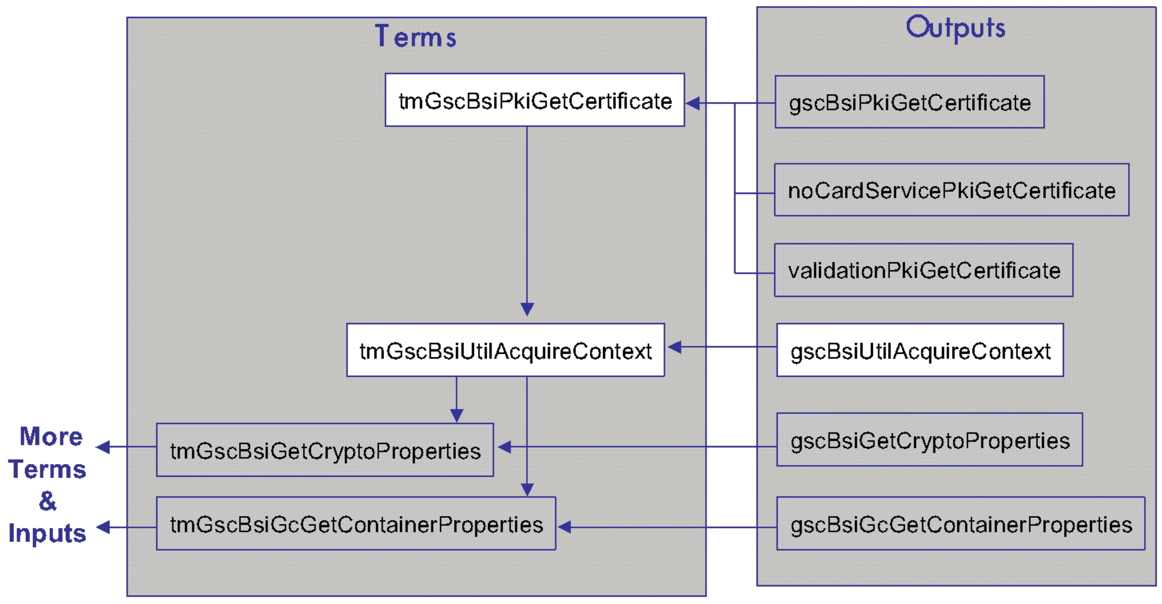
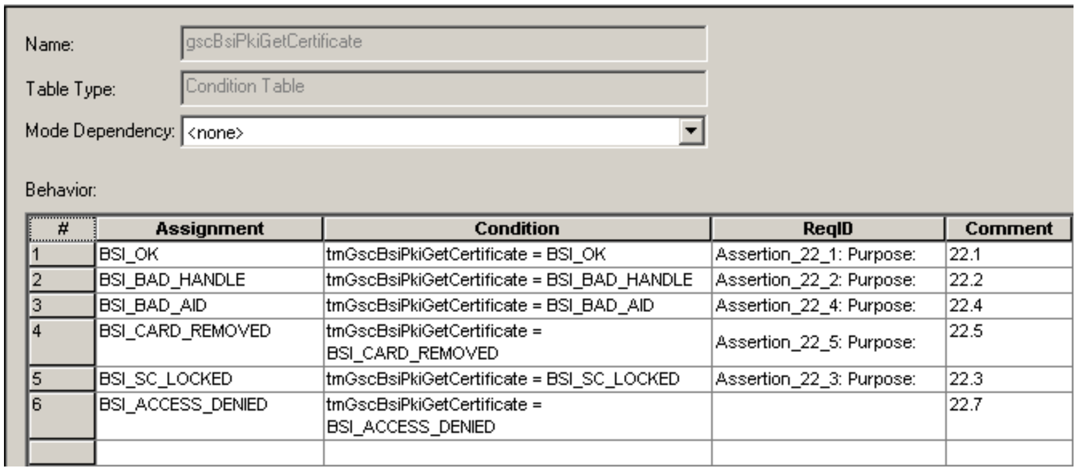

Terms
Outputs
tmGscBsiPkiGetCertificate
gscBsiPkiGetCertificate
noCardServicePkiGetCertificate
validationPkiGetCertificate
tmGscBsiUtilAcquireContext
gscBsiUtilAcquireContext
tmGscBsiGetCryptoProperties
gscBsiGetCryptoProperties
tmGscBsiGcGetContainerProperties
gscBsiGcGetContainerProperties
More Terms & Inputs

Figure 58. Dependency for gscBsiPkiGetCertificate Outputs



Name: gscBsiPkiGetCertificate

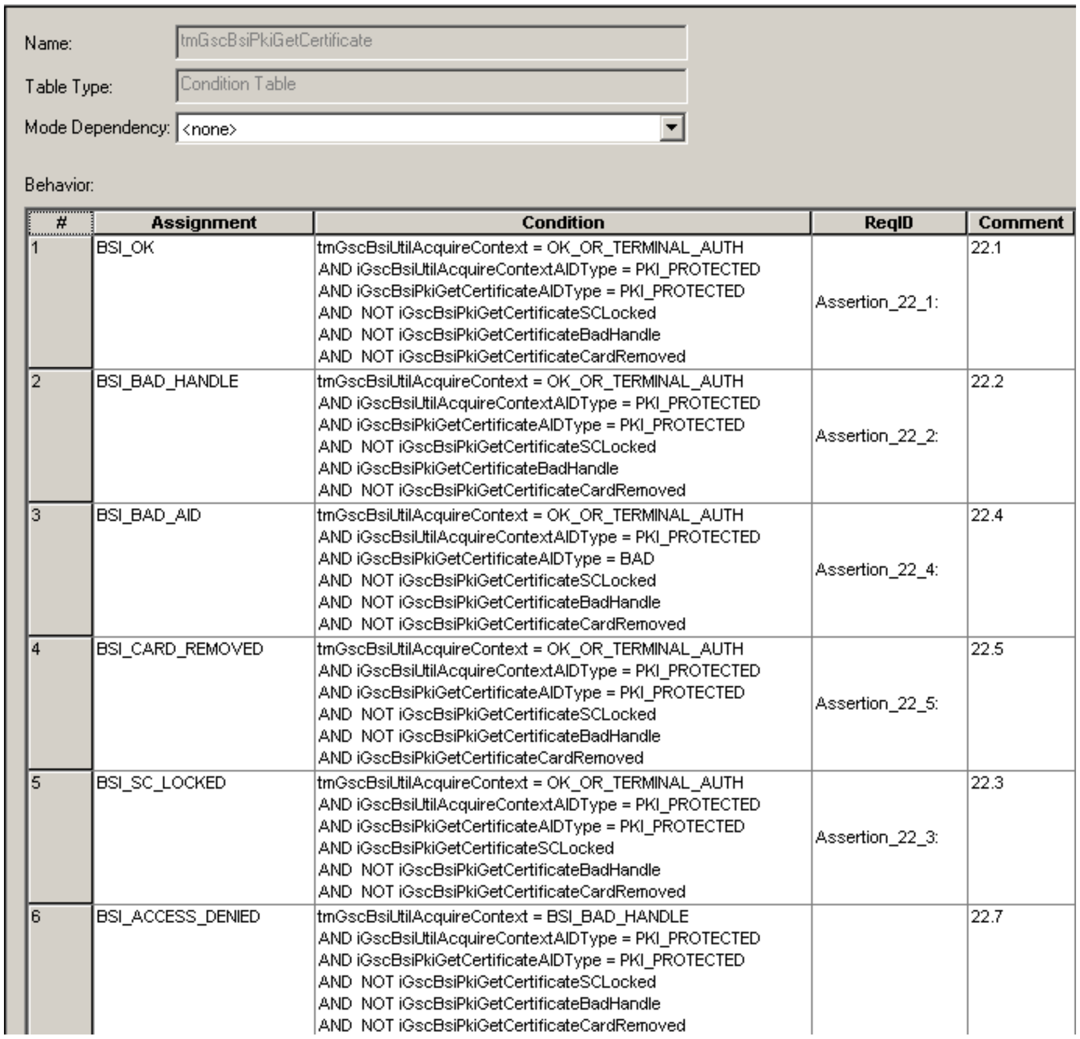
Table Type: Condition Table

Mode Dependency: <none>

Behavior:

#	Assignment	Condition	ReqID	Comment
1	BSI_OK	tmGscBsiPkiGetCertificate = BSI_OK	Assertion_22_1: Purpose:	22.1
2	BSI_BAD_HANDLE	tmGscBsiPkiGetCertificate = BSI_BAD_HANDLE	Assertion_22_2: Purpose:	22.2
3	BSI_BAD_AID	tmGscBsiPkiGetCertificate = BSI_BAD_AID	Assertion_22_4: Purpose:	22.4
4	BSI_CARD_REMOVED	tmGscBsiPkiGetCertificate = BSI_CARD_REMOVED	Assertion_22_5: Purpose:	22.5
5	BSI_SC_LOCKED	tmGscBsiPkiGetCertificate = BSI_SC_LOCKED	Assertion_22_3: Purpose:	22.3
6	BSI_ACCESS_DENIED	tmGscBsiPkiGetCertificate = BSI_ACCESS_DENIED		22.7

Figure 59. Condition Table for gscBsiPkiGetCertificate



Name: tmGscBsiPkiGetCertificate

Table Type: Condition Table

Mode Dependency: <none>

Behavior:

#	Assignment	Condition	ReqID	Comment
1	BSI_OK	tmGscBsiUtilAcquireContext = OK_OR_TERMINAL_AUTH AND iGscBsiUtilAcquireContextAIDType = PKI_PROTECTED AND iGscBsiPkiGetCertificateAIDType = PKI_PROTECTED AND NOT iGscBsiPkiGetCertificateSCLocked AND NOT iGscBsiPkiGetCertificateBadHandle AND NOT iGscBsiPkiGetCertificateCardRemoved	Assertion_22_1:	22.1
2	BSI_BAD_HANDLE	tmGscBsiUtilAcquireContext = OK_OR_TERMINAL_AUTH AND iGscBsiUtilAcquireContextAIDType = PKI_PROTECTED AND iGscBsiPkiGetCertificateAIDType = PKI_PROTECTED AND NOT iGscBsiPkiGetCertificateSCLocked AND iGscBsiPkiGetCertificateBadHandle AND NOT iGscBsiPkiGetCertificateCardRemoved	Assertion_22_2:	22.2
3	BSI_BAD_AID	tmGscBsiUtilAcquireContext = OK_OR_TERMINAL_AUTH AND iGscBsiUtilAcquireContextAIDType = PKI_PROTECTED AND iGscBsiPkiGetCertificateAIDType = BAD AND NOT iGscBsiPkiGetCertificateSCLocked AND NOT iGscBsiPkiGetCertificateBadHandle AND NOT iGscBsiPkiGetCertificateCardRemoved	Assertion_22_4:	22.4
4	BSI_CARD_REMOVED	tmGscBsiUtilAcquireContext = OK_OR_TERMINAL_AUTH AND iGscBsiUtilAcquireContextAIDType = PKI_PROTECTED AND iGscBsiPkiGetCertificateAIDType = PKI_PROTECTED AND NOT iGscBsiPkiGetCertificateSCLocked AND NOT iGscBsiPkiGetCertificateBadHandle AND iGscBsiPkiGetCertificateCardRemoved	Assertion_22_5:	22.5
5	BSI_SC_LOCKED	tmGscBsiUtilAcquireContext = OK_OR_TERMINAL_AUTH AND iGscBsiUtilAcquireContextAIDType = PKI_PROTECTED AND iGscBsiPkiGetCertificateAIDType = PKI_PROTECTED AND iGscBsiPkiGetCertificateSCLocked AND NOT iGscBsiPkiGetCertificateBadHandle AND NOT iGscBsiPkiGetCertificateCardRemoved	Assertion_22_3:	22.3
6	BSI_ACCESS_DENIED	tmGscBsiUtilAcquireContext = BSI_BAD_HANDLE AND iGscBsiUtilAcquireContextAIDType = PKI_PROTECTED AND iGscBsiPkiGetCertificateAIDType = PKI_PROTECTED AND NOT iGscBsiPkiGetCertificateSCLocked AND NOT iGscBsiPkiGetCertificateBadHandle AND NOT iGscBsiPkiGetCertificateCardRemoved		22.7

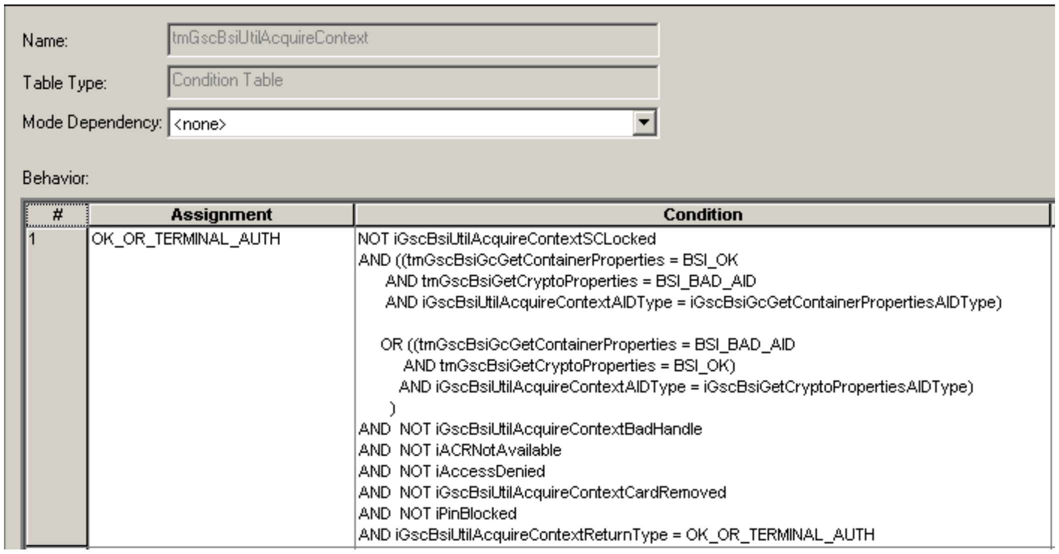
Figure 60. Condition Table for tmGscBsiPkiGetCertificate

Models that require a connection handle to the card are defined using the tmGscBsiUtilConnect term or a term that references tmGscBsiUtilConnect because this will use the gscBsiUtilConnect method to start a connection to a smart card. The model for tmGscBsiUtilAquireContext defines the condition for each return code in a manner similar to tmGscBsiUtilConnect and is defined in terms of several models as shown in Figure 61. Consider the case for BSI_OK, which is combined with terminal authentication (TERMINAL_AUTH). This scenario happens when either the container properties or the cryptographic properties have been retrieved successfully (tmGscBsiGcGetContainerProperties or tmGscBsiGetCryptoProperties). A successful reference to either tmGscBsiGcGetContainerProperties or tmGscBsiGetCryptoProperties requires a successful reference to tmGscBsiUtilConnect.

In addition, the following conditions must also hold for tmGscBsiUtilAquireContext to return OK_OR_TERMINAL_AUTH:

- An external transaction is not in progress (NOT iGscBsiUtilAcquireContextSCLocked).
- The handle to the card must be good (NOT iGscBsiUtilAcquireContextBadHandle).
- The Access Control Rules (ACRs) specified in the Authenticator are available (NOT iACRNotAvailable).
- The ACRs specified in the Authenticator are correct (NOT iAccessDenied).
- The card has not been removed (NOT iGscBsiUtilAcquireContextCardRemoved).
- The pin has not been blocked (NOT iPinBlocked).

Each different output for the model tmGscBsiUtilAquireContext defines different conditions on the inputs or terms in an attempt to define the requirements to cover all the return codes for each method.



Name: tmGscBsiUtilAcquireContext

Table Type: Condition Table

Mode Dependency: <none>

Behavior:

#	Assignment	Condition
1	OK_OR_TERMINAL_AUTH	NOT iGscBsiUtilAcquireContextSCLocked AND ((tmGscBsiGcGetContainerProperties = BSI_OK AND tmGscBsiGetCryptoProperties = BSI_BAD_AID AND iGscBsiUtilAcquireContextAIDType = iGscBsiGcGetContainerPropertiesAIDType) OR ((tmGscBsiGcGetContainerProperties = BSI_BAD_AID AND tmGscBsiGetCryptoProperties = BSI_OK) AND iGscBsiUtilAcquireContextAIDType = iGscBsiGetCryptoPropertiesAIDType)) AND NOT iGscBsiUtilAcquireContextBadHandle AND NOT iACRNotAvailable AND NOT iAccessDenied AND NOT iGscBsiUtilAcquireContextCardRemoved AND NOT iPinBlocked AND iGscBsiUtilAcquireContextReturnType = OK_OR_TERMINAL_AUTH

Figure 61. Model for tmGscBsiUtilAcquireContext

13.3.1 SMART CARD MIDDLEWARE

A smart card middleware has been developed for the NIST program to support automated test execution. For each input defined in the model (e.g., iReaderName, iCardInserted), an object mapping must be defined to cause this input to be set. An object mapping specifies the relationship between model entities and implementation interfaces that are used for sending and receiving commands from the Smart Card middleware and the card itself. As shown in Figure 62, the test middleware architecture enables generated test vectors to be injected into the smart card middleware/on-card implementation for conformance testing to GSC-IS. It controls setup, execution, and cleanup after each test. In addition, it provides other logging and reporting capabilities that are important for fully automated test execution and failure analysis.

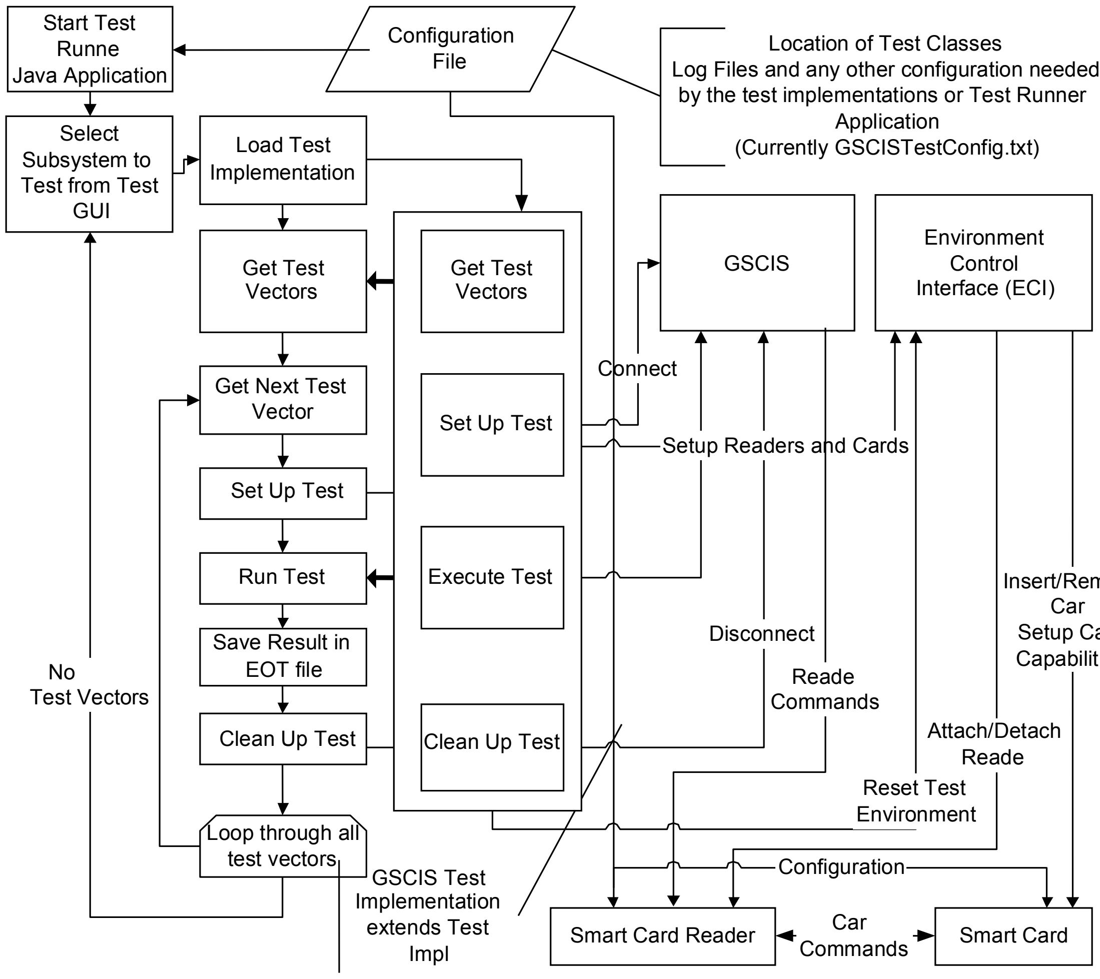

Start Test Runne Java Application
Configuration File
Location of Test Classes Log Files and any other configuration needed by the test implementations or Test Runner Application (Currently GSCISTestConfig.txt)
Select Subsystem to Test from Test GUI
Load Test Implementation
Get Test Vectors
Get Test Vectors
GSCIS
Environment Control Interface (ECI)
Get Next Test Vector
Set Up Test
Connect
Set Up Test
Setup Readers and Cards
Run Test
Execute Test
Save Result in EOT file
Clean Up Test
Clean Up Test
No Test Vectors
Disconnect
Reade Commands
Insert/Remove Car Setup Card Capabilities
Attach/Detach Reade
Reset Test Environment
Loop through all test vectors
Configuration
GSCIS Test Implementation extends Test Impl
Smart Card Reader
Car Commands
Smart Card

Figure 62. Smart Card Middleware for Automated Test Execution

A GSC-IS implementation must interact with a card reader driver, a smart card reader, and a smart card. To test a GSC-IS implementation, the setup of the smart card must be modified by the test driver script for each test cycle. Operations such as attaching and detaching readers and setting the card capabilities cannot be performed through the GSC-IS. The ECI is used to provide interaction with the external environment, connect card readers, and insert/remove cards from the readers. The ECI is implemented in two ways:

- Manual intervention
- Software-controlled automation of the test environment

A manual intervention version of the ECI prompts a user to perform operations at appropriate points in a test cycle, such as:

- Reader attachment
- Smart card setup
- Smart card insertion/removal

This manual intervention supports testing hardware implementations of the GSC-IS. This approach, however, has the potential for introducing human error into the test results. Therefore, the desired approach is to use software-controlled automation. A manual intervention version of the ECI is implemented and can be used when testing any hardware implementation. Software-controlled automation of the test environment requires that a custom implementation of the ECI be built. A custom version of the ECI interacts with the simulator software implementations of the card reader and smart card. The simulator ECI implementation does not require the manual intervention required by the default ECI implementation designed for testing hardware.

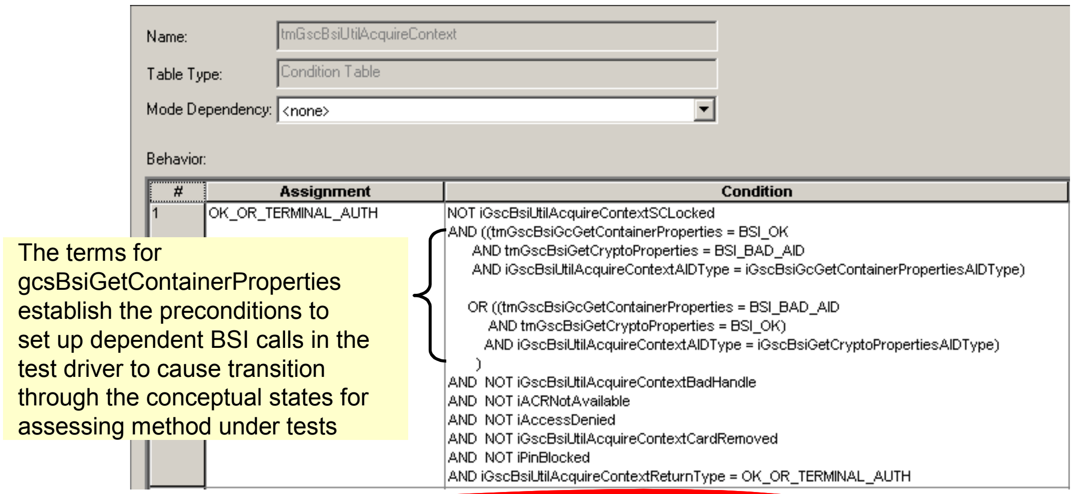
13.4 KEY GUIDELINES

13.4.1 TEST INFRASTRUCTURE

Development a test infrastructure (referred to by NIST as a test middleware) that establishes a common basis for test injects, ensures proper initialization and controls, and fosters more robust approaches to support test automation. Use the test infrastructure to support common reporting, logging, and measurements. In working on other programs with companies, a test infrastructure promotes design for testability, which is essential for test automation. For programs that are based on product families or will evolve through many versions, a test infrastructure can be cost-effective, because it can be reused.

13.4.2 MODEL DEPENDENCIES AND TEST SEQUENCING

Analyze and model based on dependency relationships of component functions to promote reuse of common models and associated test sequencings. For example, the conceptual state machine shown in Figure 55, with a partial representation in Figure 63 relates the sequence of BSI method calls to achieve the Context Acquired state that is associated with the gcsBsiUtilAcquireContext BSI method call. The term tmGscBsiGcGetContainerProperties is defined as a precondition in the condition table. Its test driver mapping establishes the set of calls to get to the state Has Container Properties. Therefore, the dependencies in the term relationships directly support the sequence of calls to the smart card to transition an application to the various states. In addition define the object mappings in one place and reuse them for each related model's tests.



Name: tmGscBsiUtilAcquireContext

Table Type: Condition Table

Mode Dependency: <none>

Behavior:

#	Assignment	Condition
1	OK_OR_TERMINAL_AUTH	NOT iGscBsiUtilAcquireContextSCLocked AND ((tmGscBsiGcGetContainerProperties = BSI_OK AND tmGscBsiGetCryptoProperties = BSI_BAD_AID AND iGscBsiUtilAcquireContextAIDType = iGscBsiGcGetContainerPropertiesAIDType) OR ((tmGscBsiGcGetContainerProperties = BSI_BAD_AID AND tmGscBsiGetCryptoProperties = BSI_OK) AND iGscBsiUtilAcquireContextAIDType = iGscBsiGetCryptoPropertiesAIDType)) AND NOT iGscBsiUtilAcquireContextBadHandle AND NOT iACRNotAvailable AND NOT iAccessDenied AND NOT iGscBsiUtilAcquireContextCardRemoved AND NOT iPinBlocked AND iGscBsiUtilAcquireContextReturnType = OK_OR_TERMINAL_AUTH

The terms for gcsBsiGetContainerProperties establish the preconditions to set up dependent BSI calls in the test driver to cause transition through the conceptual states for assessing method under tests

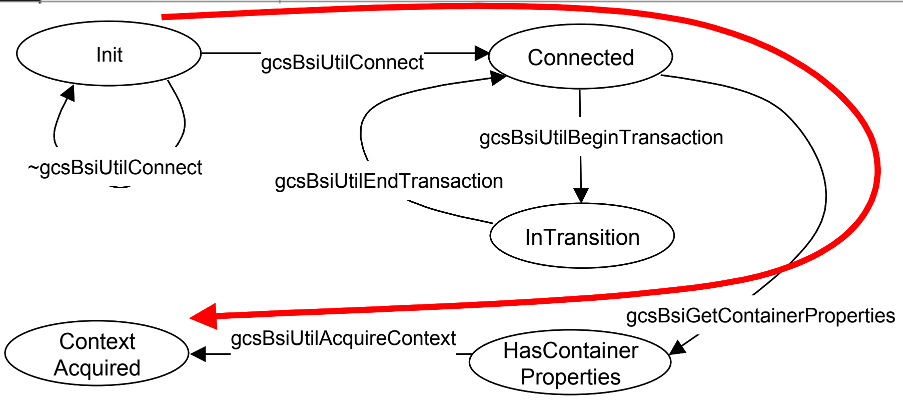
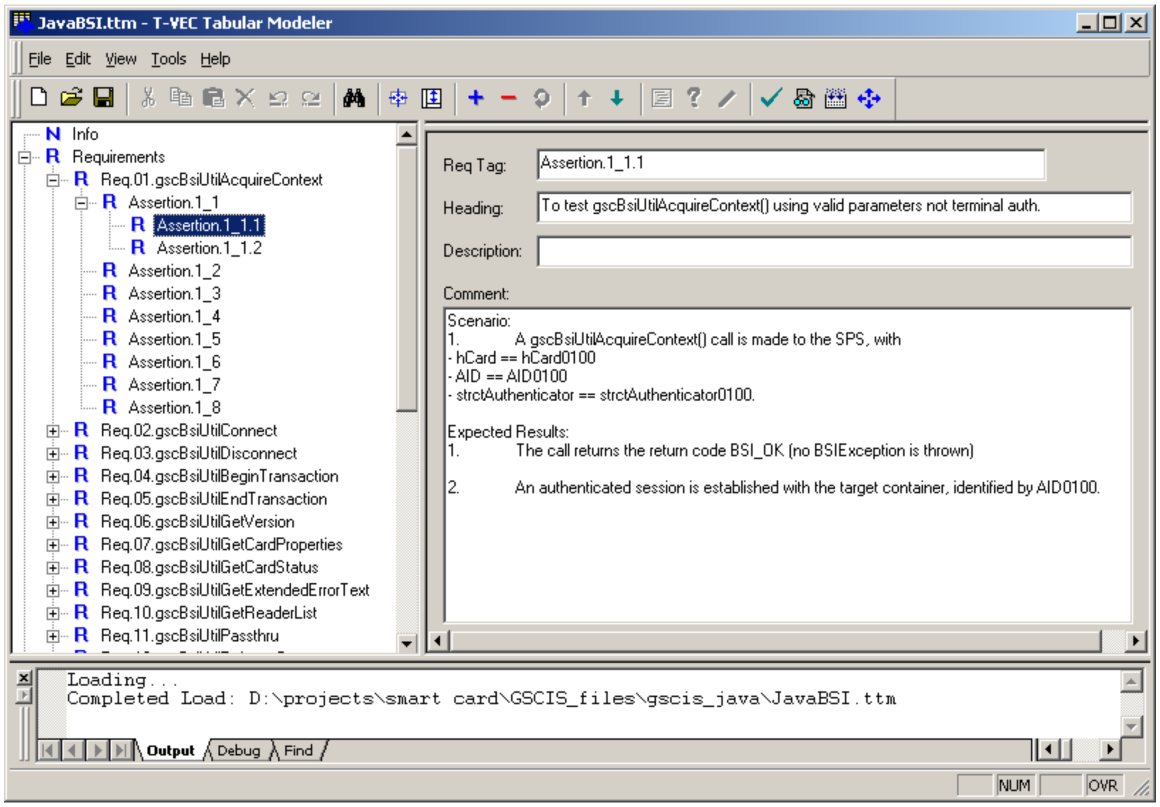


Figure 63. Partial State Machine for Context Acquired

13.4.3 Requirement Traceability

There are many reasons to use requirement traceability links to the model. In particular, it permits test failures to be traced back to the requirements. In the case of the smart card example, an independent test group provided a set of test assertions that they presumed would be the minimal set required to cover the functionality of the specification, as shown in Figure 64. Specifying requirements, or in this case test assertions, provides a basis for assessing the completeness of the model. As shown in

, each assertion was linked to a modeled requirement, but there were some cases modeled that did not have test assertion (e.g., for the assignment of BSI_ACCESS_DENIED). This case points out that the modeling process tends to be more thorough because it captures relationships associated with all values of the interfaces, even if they are not specified within the documentation or test assertions.


JavaBSI.ttm - T-VEC Tabular Modeler
File Edit View Tools Help
Output
Debug
Find
NUM
OVR

N Info
R Requirements
R Req.01.gscBsiUtilAcquireContext
R Assertion.1_1
R Assertion.1_1.1
R Assertion.1_1.2
R Assertion.1_2
R Assertion.1_3
R Assertion.1_4
R Assertion.1_5
R Assertion.1_6
R Assertion.1_7
R Assertion.1_8
R Req.02.gscBsiUtilConnect
R Req.03.gscBsiUtilDisconnect
R Req.04.gscBsiUtilBeginTransaction
R Req.05.gscBsiUtilEndTransaction
R Req.06.gscBsiUtilGetVersion
R Req.07.gscBsiUtilGetCardProperties
R Req.08.gscBsiUtilGetCardStatus
R Req.09.gscBsiUtilGetExtendedErrorText
R Req.10.gscBsiUtilGetReaderList
R Req.11.gscBsiUtilPassthru

Req Tag: Assertion.1_1.1

Heading: To test gscBsiUtilAcquireContext() using valid parameters not terminal auth.

Description:

Comment:

Scenario:
1. A gscBsiUtilAcquireContext() call is made to the SPS, with
- hCard == hCard0100
- AID == AID0100
- strctAuthenticator == strctAuthenticator0100.

Expected Results:
1. The call returns the return code BSI_OK (no BSIException is thrown)

2. An authenticated session is established with the target container, identified by AID0100.

```
Loading...
Completed Load: D:\projects\smart card\GSCIS_files\gscis_java\JavaBSI.ttm
```

Figure 64. Smart Card Requirement Traceability

13.5 RESULTS

This case study describes the model-based development and test generation method used for creating models for the BSI of the GSC-IS. The model for 23 BSI methods resulted in 306 requirement threads that have been used to produce 692 test cases, without using inlines. (See Section 13.4] for more information on inlining.) The case study also provides a summary of the test middleware that supports automated conformance test execution of the Java language bindings for the BSI services. The middleware provides the environment to support a common approach to perform all tests.

This case study illustrates a complete end-to-end use of model-based testing. Starting from a well-defined and well-reviewed specification, models were developed. The detailed analysis that modeling forces on the requirements process exposed anomalies that had gone undiscovered during all the review processes for the different versions of the GSC-IS specification. The most notable result was that the TAF team uncovered 20 significant issues and several minor issues with version 2.1 of the GSC-IS specification. These issues were reported to NIST. The structural nature of the modeling process supports defect identification.

The requirement traceability process is useful for tracing requirements to tests, but this case study illustrates how incompletenesses in the requirements can be exposed. The test infrastructure illustrates a robust and reusable mechanism to automate all the test cases, test sequences, and test results capture, while supporting test logging and reporting.

NIST was extremely pleased with the results, and this infrastructure is the basis for the latest effort resulting from the Homeland Security Presidential Directive HSPD-12 [http://www.fas.org/irp/offdocs/nspd/hspd-12.html] that calls for new standards to be adopted governing the interoperable use of identity credentials to allow physical and logical access to federal government locations and systems. The Personal Identity Verification (PIV) for Federal Employees and Contractors, Federal Information Processing Standard 201 (FIPS 201 [http://www.csrc.nist.gov/publications/fips/fips201/FIPS-201-022505.pdf]) establishes standards for identity credentials. The Special Publication 800-73 (SP 800-73 [http://csrc.nist.gov/publications/nistpubs/800-73/SP800-73-Final.pdf]) specifies interface requirements for retrieving and using data from the PIV Card and is a companion document to FIPS 201. The approach used for modeling and testing the smart cards is being applied to the PIV specification and associated card implementations.

14. MEDICAL DEVICE PRODUCT LINE

The information presented in this section is generic. The TAF team examined several different companies' technical specifications from Internet information and patent summaries to ensure the following information is presented in a product-neutral form. Several companies have similar products with conceptually similar processes as discussed in this section.

14.1 PROBLEM

Like many companies that build high-assurance systems for life-critical applications, zero defects is a requirement. The cost of the V&V efforts for these companies often exceed 50% of the total effort, and the company discussed in this case study did confirm that its testing cost was significantly higher than 50% of the life-cycle cost. In addition, the complexity of its systems continues to increase, along with greater scrutiny from the certification authorities such as the FDA, and competitive market pressure means these high-assurance requirements must be satisfied in even shorter time periods.

This company has advanced testing facilities including simulators, emulations, breadboard and hardware test environments, with comprehensive test analysis, measurement, tracking, reporting, and logging capabilities. It is desirable to reduce the cost of testing, but schedule reduction is the most critical need in order to remain competitive in the marketplace. Most testing prior to the use of TAF has been performed using manually produced test scripts that support fully automated test execution and results analysis. Even with this significant V&V support, creating test cases (i.e., the test design process) and implementing those test designs into scripting languages have become labor-intensive, time-consuming, and costly tasks. The criticality of the systems requires them to perform comprehensive reviews of test procedures that can be several hundred lines of code. For any small product, there can be over one thousand test scripts required to fully verify the product.

For changes made to a device after it has been released for clinical trials or to the field, regression testing requires that the entire test suite must be reexecuted. Often, because complex timing requirements require test scenarios to simulate the human anatomy, test scripts that might work for one release of a product might not work after a modification has been made to the system. Such tests must be reassessed, corrected or re-implemented, rereviewed, and then reexecuted. If common changes are required, such changes could require reediting of hundreds of test scripts.

A comprehensive simulation and test infrastructure provides significant advantages. However, the sophisticated and wide-spectrum set of APIs for controlling human simulations sometimes provides far too many options for test designers and can lead to reduced robustness of the test scripts, especially since different simulation APIs have different timing characteristics—that is, the timing of one sequence of API calls can vary by a few milliseconds from another set, even though they might achieve the same function.

Currently, receiving FDA approval is a critical and sometimes time-consuming part of product release. If the FDA can be convinced that the TAF approach provides the verification rigor needed for FDA certification and will allow the TAF verification results to be submitted as justification for approval, then the company will be able to improve its ability to deliver complex systems with FDA approval in a more cost-effective manner.

This case study discusses the TAF team's involvement with a company over a multiyear time period to create an engineering approach to model-based testing. The team worked with the system engineers that develop the requirement and interface specifications, the design team that constructs more testable system and components, the test engineering organization, the quality assurance organization that interfaces with the FDA, and the organization that develops and maintains that advance engineering infrastructure.

14.2 APPROACH

The effort started with a small thread of functionality and transitioned into one of the most complex control mechanisms that is common in many devices. These successful demonstrations lead to full-scale development of two different product lines, and involved coordination with the design team, system engineering team that wrote the product technical specification, test team, and the quality assurance organization involved in FDA certification and tool qualification. The process and infrastructure was used in a fully tailored custom training class based on one of the verified products.

The modeling process was consistent with the process picture as reflected in Figure 16. In addition, the modeling started nearly in parallel to the design and implementation process. This approach permitted more continuous testing during development and allowed for early analysis of the technical specifications. This case study provides details related to organizing models to support multiteam development and other related benefits.

Many of the devices this company develops tend to have a common data flow as conceptually represented in Figure 65. In real time on a periodic basis, the devices usually perform some sensing of a heart while capturing information and check that information against stored information within the device that is usually set by a doctor. Based on the information, usually collected and filtered over time, algorithms select options to issue a therapy, such as increasing or decreasing a pace, or optimizing the device to preserve battery life. These devices continue to evolve over time, and some doctors prefer different algorithms. It is common for functions such as Check to have many different types and combination of filtering, matching, and selection possibilities that suit different doctors' views on patient treatments. A new combination often is called a feature when it is presented to a doctor; however, the feature can impact many components within a system. This case study discusses the organizational and process impacts of developing a feature for the Check component that impacts Filter, Match, and Select.

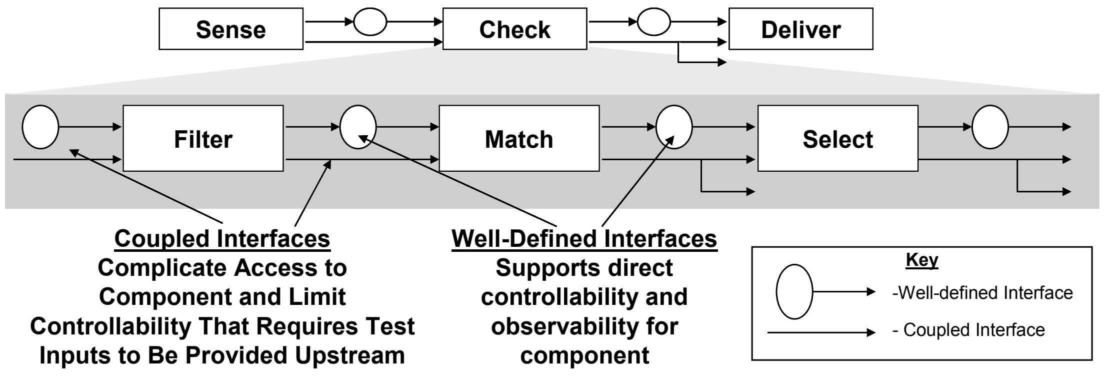

Sense
Check
Deliver
Filter
Match
Select
Coupled Interfaces Complicate Access to Component and Limit Controllability That Requires Test Inputs to Be Provided Upstream
Well-Defined Interfaces Supports direct controllability and observability for component
Key
-Well-defined Interface
- Coupled Interface

Figure 65. Conceptual Components of Example Medical Device System

This case study takes a chronological perspective because the integration of the entire model-based method impacted many different organizations within this company. For example, prior to the engagement with the TAF team, as reflected in Figure 65, the components of the Check function were not partitioned with well-defined interfaces; rather, the functionality was coupled, which made testing the functionality in each subcomponent (i.e., Filter, Match, and Select) more difficult. However, there is a verification requirement to demonstrate that every thread through a component or subcomponent is completely tested. Tight coupling makes this requirement difficult to achieve and demonstrate.

14.3 IMPLEMENTATION

There were four distinct phases of involvement with this company. Phase I was essentially a short pilot project effort to demonstrate the feasibility of applying model-based testing. During this phase, the TAF team quickly (i.e., approximately 2 hours) developed a model, mapped test drivers to the test environment, executed a test against a project, and found a minor problem with the memory mapping for the uploaded message from the device. These achievements encouraged the company to progress to Phase II.

14.3.1 IMPROVED TEST INFRASTRUCTURE

Phase II was a challenging problem because the model characterized a well-defined but arguably one of the most complex control mechanisms of the entire device. The team modeled approximately 130 requirement threads. The modeling process helped illustrate problems and anomalies, nothing serious, with documentation, including the technical specification and interface specifications, which were maintained separately. However, the key issues arose when the team created the object mappings to support automated test driver generation.

The test infrastructure was robust and used by both testers and developers. Test scripts were written in C++ in a Microsoft development environment, and an extensive set of API services provided numerous ways to integrate different software versions with a simulation for the heart. The simulation services permitted program control for such things as the heart rate to ventricle and atrial timing, but unfortunately, the API services had many overlapping functions. It was often difficult to understand how to properly initialize the simulation for a particular subsystem of related features; in addition, it was difficult to uncover the particular functions necessary to set up consistent control of the simulator. These same problems plagued test engineers, especially less experienced engineers, because there were several hundred API services to support testing and many different requirements for initializing different test environments, for various types of product features.

The team worked with some knowledgeable people that helped develop the test infrastructure and simulator to successfully build test drivers for the modeled requirements. The team isolated the best services to accomplish the task and used those to produce the generated test drivers. More importantly, the efforts caused an initiative within the company to redesign the entire set of API services for the user community. The developers simplified the set of APIs and reduced them to about one quarter of the original number.

14.3.2 Requirement Analysis

The success of Phase II permitted the team to move on to a new feature for an existing device, which is referred to as Phase III. Originally, this feature was going to be included in the next-generation device, but market pressure forced this company to include this feature in an existing device.

The team recommended the interface-driven requirement modeling that starts early during the requirement and design phase. This early modeling has been demonstrated to help create a more testable design and improve the requirement and interface specifications.

This company uses a two-phased approach to the release of a technical requirement specification. During first phase, the technical specification is under configuration control but can be evolved, reviewed, and changed without official approval from a change control board. After the specification has been released, a change control board must approve all changes.

Fortunately, the modeling process started from a technical specification that had not been officially released. This was an exception to the typical testing process because testing normally starts much later in the development process. However, during the process of modeling the requirement specification, about 100 specification problems were uncovered. Prior to the change control board, all these issues were discussed and resolved with the system engineers that developed the technical specification. This intangible benefit of the model-based testing effort saved cost and effort by uncovering these issues. Early resolution of the issues also saved the designers time and effort in making potentially bad design choices because of issues in the requirement specification.

14.3.3 Design for Testability

Another issue that surfaced during Phase II was addressed during Phase III. As reflected in Figure 65, the functionality in the existing system was tightly coupled because of numerous historical reasons related to power consumption and memory space limitations of the device. The interfaces between Filter, Match, and Select were not well-defined. This complicated the testing process, requiring many tests to be initiated from higher levels in the system, such as Check, because some of the inputs could be set upstream from the Check component. In addition, the outputs from the function such as Match were not visible. This made systematic and comprehensive testing of these lower-level components difficult. Normally, ensuring coverage of the threads through the implementation of these lower-level components means increased testing from the high-level components. Sometimes, the number of tests can increase by an order of magnitude.

The team started this effort early enough that the designers were able to expose the interfaces of both the inputs and outputs, including internal state information to increase the testability significantly. Approximately 80% of the functionality was tested with improved interface support provided by the design team. This approach significantly reduced the complexity of the model and the tests, and provided greater test coverage with fewer tests to reduce time and cost. The remaining 20% represented elements of the components that could not be changed due to performance issues, and impacts on cost, resources, and schedule associated with retesting.

The team applied this design-for-testability philosophy to another product in Phase IV. The Phase IV effort involved an older product that was being replaced by a new product. Again, because of market pressure, the company decided to add a feature to an older product. The success on the Phase III effort provided substantial evidence for repeating the effort on Phase IV.

14.3.4 Modular Requirement Specifications

This company has some of the best technical specifications and interface documentation of any member company; however, during the modeling process, the team identified a justified reason for organizing the specifications in a different way. As shown in Figure 66, the company uses an interface specification that is a separate document from the requirement specification. Although this is oversimplified, conceptually one team member specified a model for the requirements in Section 1.1. A second team member made the model for Section 1.2, and another made a model for Section 1.3 of the requirement specification. The issue that emerged during the modeling process of Section 1.4, which describes feature interaction requirements between Filter, Match, and Select, is that many of these features described conditions that were already defined in a model. Because these modeled requirements had gone through the review process, and all the tests generated from these models were complete, with passing status, the team decided decision to develop a separate model for the requirements of Section 1.4.

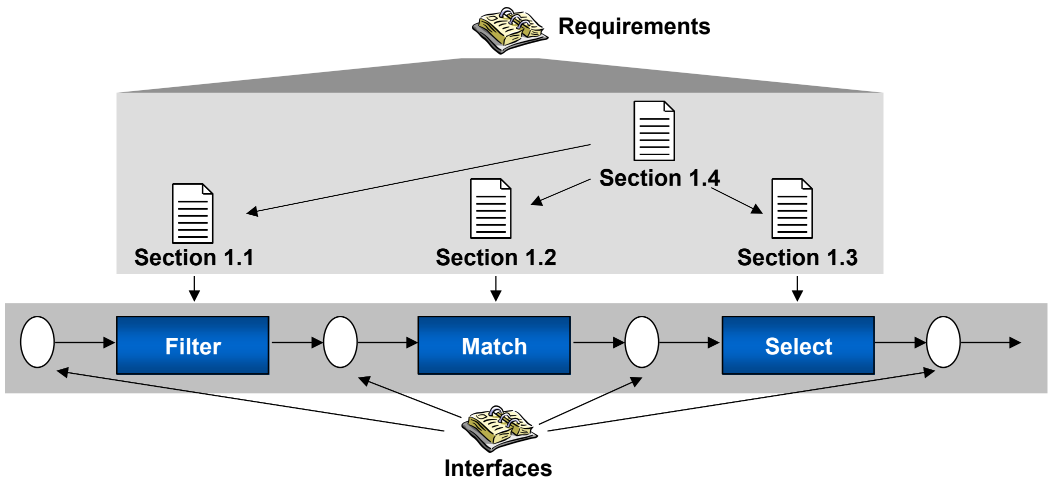


Figure 66. Organization of Requirement Feature

The model-include mechanism allowed the Feature Interactions model, associated with Section 1.4, to include the Filter, Match, and Select models so that existing terms could be reused, as shown in Figure 67. If conditions change in the future, the changes can and should be made in a single place. Just as it is a good practice to separate the interface specification in code (e.g., in a .h include file for the C programming language), it is a good practice to specify component interfaces separately. Figure 67 illustrates how common models represented interfaces separate from the required behavior. If the interface is related to the requirements, the interface model can be included with the model behavior. This practice is important because, if the interface changes, the changed interface is isolated in one place. Interface models tend to correspond with object mappings that represent the interface to the implementation. Section 14.3.6) provides more details on common object mappings and test infrastructure.

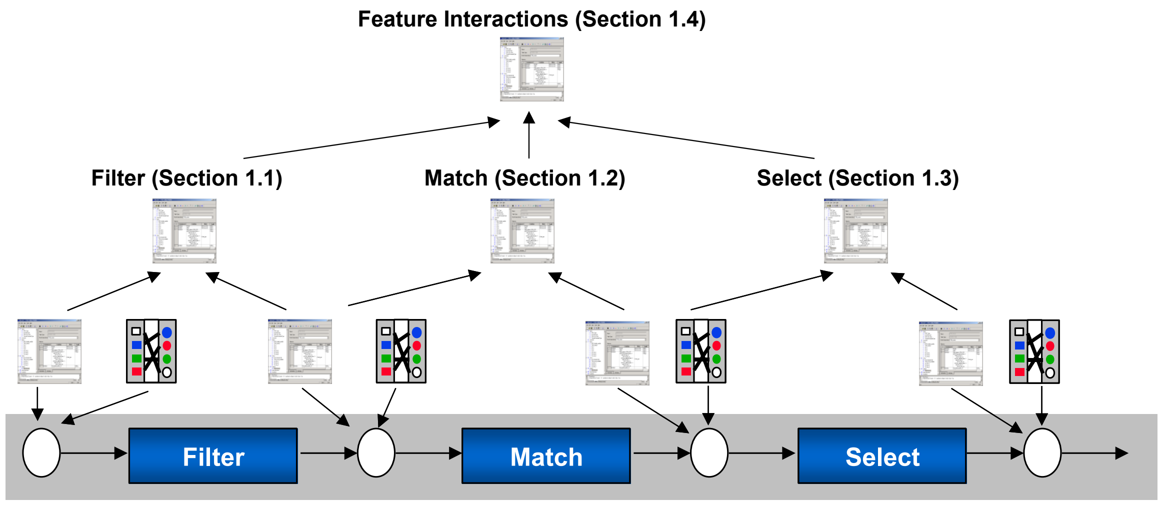


Figure 67. Models Represent Interfaces and Required Behavior

An important guideline, pointed out to the system engineers and specification team, is that the specification of the requirements should be associated with outputs used to assess the verification of the requirements. That means the requirement on interactions between components may have been better specified in the appropriate sections in Sections 1.1 through 1.3 as they related to the interfaces of Filter, Match, and Select.

14.3.5 Model-Based Review Process

Companies that develop safety-critical applications often are required to have code reviews as well as test procedure reviews. This company's existing process often required reviews of hundreds of test scripts that may have hundreds of lines of code. The new process relied on using validated TAF tools that satisfied the quality assurance organization's criteria for proper tool operation; this meant that the quality assurance organization believed that the tools would produce test vectors and test drivers that were complete and correct with respect to the model of the requirements. This permitted the review process to change.

The new process involved a review of the model by the system engineers to ensure the completeness and correctness with respect to the requirements. In addition, all requirements used the requirement traceability mechanism illustrated in Figure 9. The second part of the process involved the designer/implementers, who reviewed the models and the associated test vectors presented in matrix format also as illustrated in Figure 9. The review of the model for the associated requirements that were directly traced to the requirements was much easier to understand and verify than the test drivers for the model.

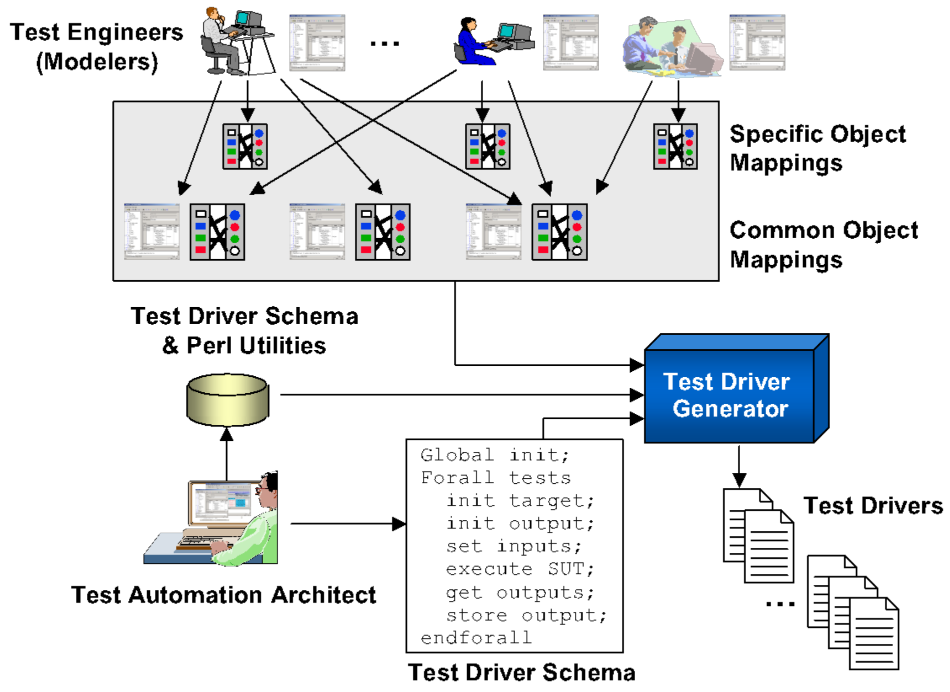
Design decisions, implemented in code, result often in undocumented, implementation-derived requirements. These implementation-derived requirements must be tested too. An important addition of the TAF process is that the designers were able to request a special type of model information called a test constraint for the implementation-derived requirements. A test constraint results in tests, in addition to the requirement-based tests, to support implementation-derived requirements. This approach further reduces the unit testing effort typically performed by the developer. Because tests were now being run in parallel to development, the implementers work effort was reduced. This reduction in work, however, would not have been possible without the designer providing additional test interfaces at the lower-level components.

The early interaction between the designers and test engineers improved the interfaces for testability, provided continuous testing earlier to reduce unit testing by the designers/implementers, and reduced the complexity of the testing to achieve more comprehensive test coverage with a reduced set of model-based tests.

14.3.6 Multiteam Model and Test Infrastructure

There are often significant skill and knowledge differences within an organization, and as discussed in previous sections, the TAF team relied on knowledgeable individuals to recommend simulation API services for scripting tests and for initializing the test environment in order to automate test execution. These knowledgeable individuals were able to recommend specific services to carry out the functions to control the simulation. The test infrastructure for model-based testing can be engineered to provide significant reuse of model interfaces and their associated object mappings by leveraging the most knowledgeable resources within a company.

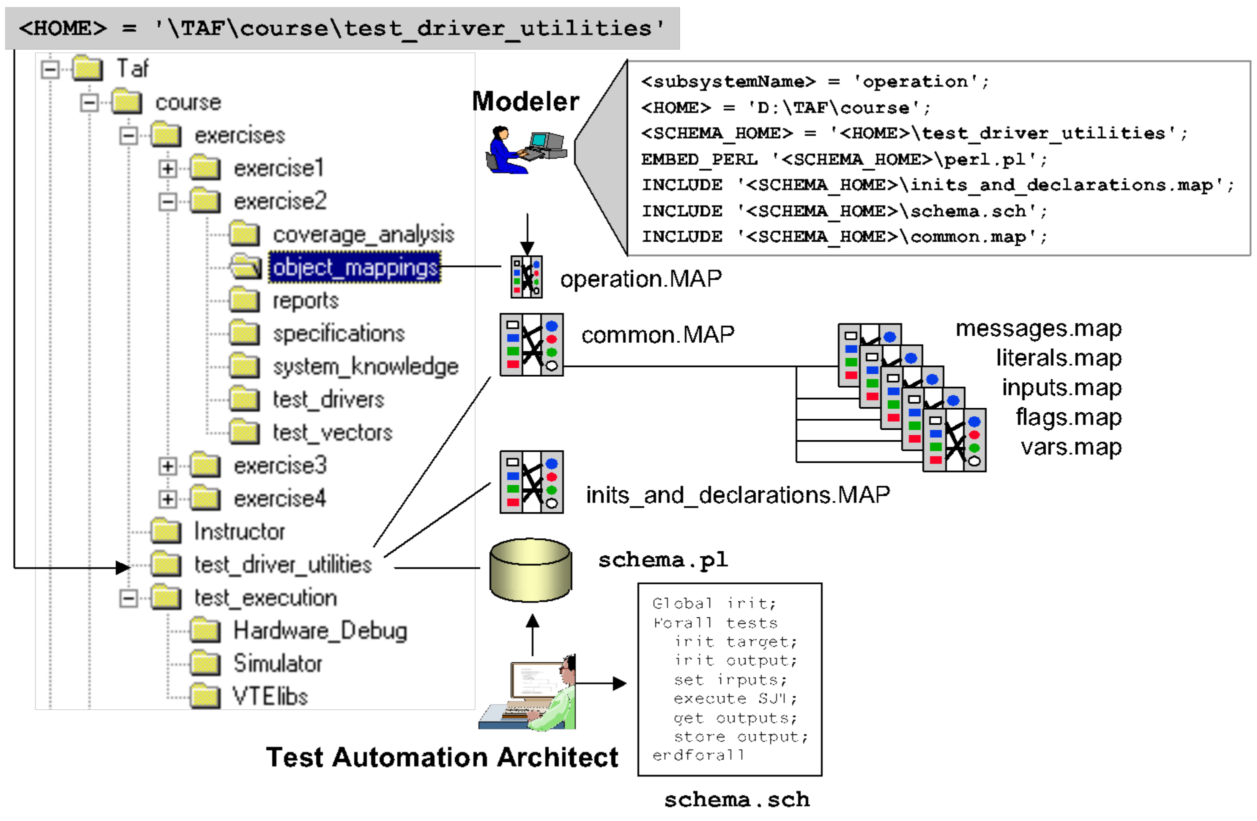
Figure 68 illustrates the two roles involved in the back-end aspects of model-based testing. This is the process where modeled variables more closely related to the requirements must be mapped to the physical mechanisms that are used to set inputs (i.e., inject test inputs) and get outputs. The two roles include the Modeler and the Test Automation Architecture (who often plays a modeling role too). In this company, there was one test automation architect for all of the modelers supporting the Phase III and Phase IV effort. This one individual tailored and evolved the test driver schema to operate with two completely different testing environments and languages. The schema provided common reporting and execution mechanisms all based on the same framework, which is shown in Figure 69. The test automation architecture also controls the common object mappings that correspond to the common component interfaces. A modeler that might not know as much about the details on the test environment can focus on building the models from the requirements and then be directed modeling lead to reference common object mappings in order to produce test drivers for all of the modeled functionality that works with the concrete interfaces of the actual target systems or simulation.



```
Global init;
Forall tests
   init target;
   init output;
   set inputs;
   execute SUT;
   get outputs;
   store output;
endforall
```

Figure 68. Roles in Test Driver Development

As shown in Figure 69 a modeler must define one object mapping for each output defined in the model. Within the object mapping, references are made to directory paths such <HOME> that references the example path for the directory of the project (e.g., D:\TAF\course). From that <HOME> user-defined variable, other information is referenced, such as the location of the schema (e.g., <SCHEMA_HOME> = '<HOME>\test_driver_utilities'), which is where the test automation architect provides common Perl scripting utilities for reporting and logging, common initializations, and declarations related to initializing the test environment, along with a common schema and common mapping file. As reflected in Figure 69, the common mapping file (i.e., common.map) includes other common information that is pertinent to all modelers, such as messages, literals, inputs, flags, and other variables. If an interface changes for some input, it is changed in one location (i.e., the inputs.map object mapping file), and all models that reference that input variable have a test driver interface that uses the new access method for setting that input variable. When such a change occurs, all test drivers can be regenerated to use this new test interface. Ensuring object mapping information is defined in one place avoids the problem in the current approach were every test script that references the input variable would need to be modified manually through some type of editing process.



```
<HOME> = '\TAF\course\test_driver_utilities'
```

```
<subsystemName> = 'operation';
<HOME> = 'D:\TAF\course';
<SCHEMA_HOME> = '<HOME>\test_driver_utilities';
EMBED_PERL '<SCHEMA_HOME>\perl.pl';
INCLUDE '<SCHEMA_HOME>\inits_and_declarations.map';
INCLUDE '<SCHEMA_HOME>\schema.sch';
INCLUDE '<SCHEMA_HOME>\common.map';
```

```
Global init;
Forall tests
   init target;
   init output;
   set inputs;
   execute SUT;
   get outputs;
   store output;
endforall
```

Figure 69. Test Infrastructure Organization

During both Phase III and Phase IV, the simulation environment, test infrastructure, component interfaces, and reporting requirement continued to evolve. The test automation architect, through updates to common object mappings, the schema, or Perl utilities, managed all these changes in a way that was transparent to the other modelers.

14.3.7 Model-Based Measurement

During the beginning of Phase IV, which was about the middle of the Phase III project, the team recognized that they had TAF measurement information to support project measurement. See *Guidelines for Using Test Automation Framework Measures* [Consortium 2004] for more information.

The team developed a consolidated spreadsheet to track key measures that helped predict project completion information. Figure 70 provides a perspective on the key measurement information and how it relates to the typical TAF method of interface-driven requirements modeling. With this approach, there are four key base measures. See Appendix 0 for more details. System engineers are responsible for producing requirements, which results in the base measure number of requirements. A test engineer or modeler works in parallel with developers to refine requirements and build models to support iterative testing and development. Modeling introduces model variables, which results in the base measure number of variables. After model translation and processing, the model requirements are converted into DCPs, which is a base measure related to requirements. To support test driver generation and test execution and results analysis, the base measure number of object mappings is used. Object mappings relate model variables to the implementation interfaces.

This measurement-related information helped managers and project leads with predicting schedule duration and estimating project completion dates. Historical measurement information can be used prior to the start of a project, but it also is important to use data derived during the project.

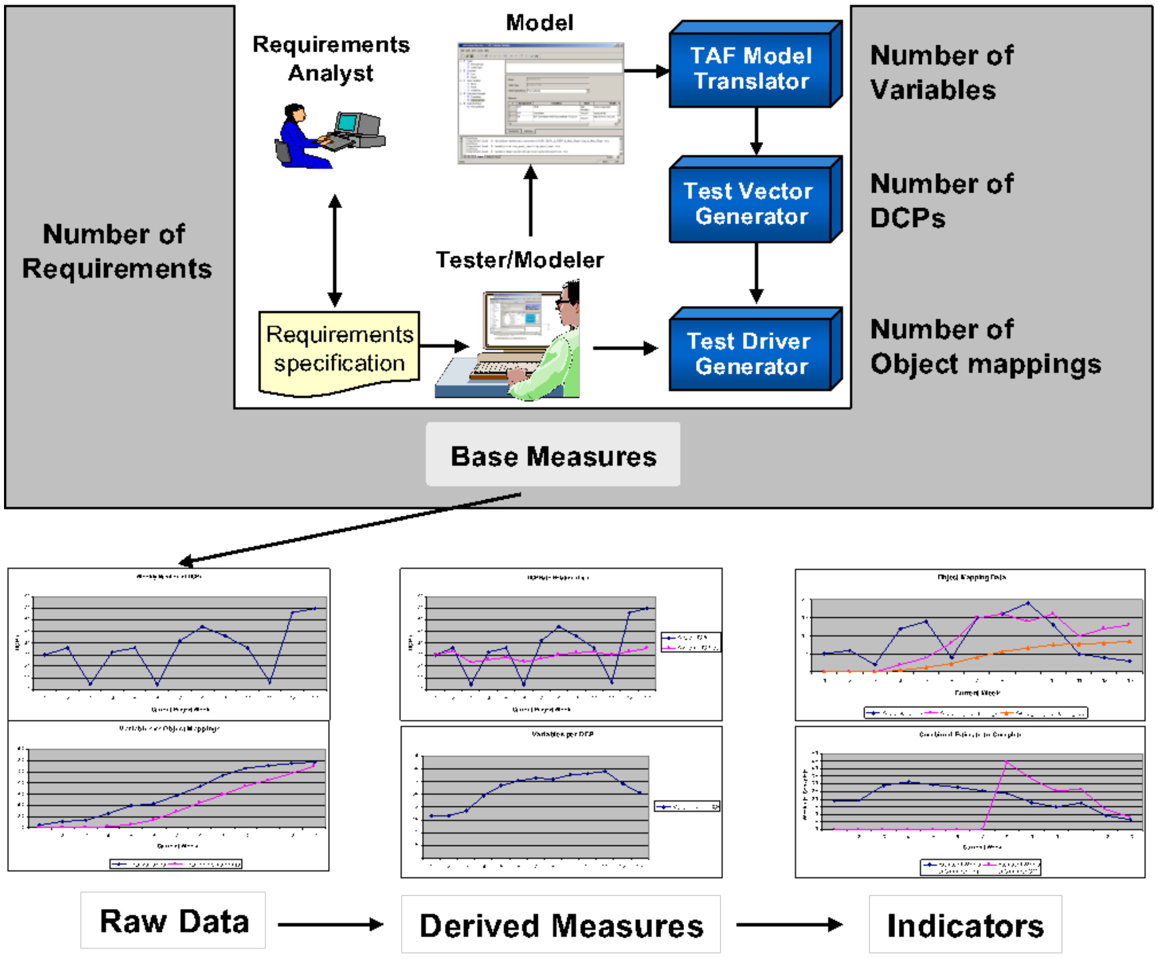


Figure 70. Process View of TAF Measurement

14.3.8 Configuration Control

For high-assurance systems, the certification authorities such as the FDA require all test artifacts to be configuration controlled. One key reason is that if some problem in a particular release occurs in the future, configuration controlled artifacts, such as test scripts, can be reviewed to assess potential deficiencies in processes such as test procedure design, test script initializations, or test reviews.

Companies often ask the TAF team how the process of configuration management changes when transitioning from manual test design and execution to model-based testing. Figure 71 reflects the change in the process for this company (the actual artifacts are named differently).

The previous process involved four types of artifacts. One or more test requirements matrices that captured the traceability of the requirement specification map to one or more test designs stored as text files. A test design often includes information about decomposition of the requirements into test requirements that were to be tested by one or more test cases. Each test design had a one-to-one correspondence to a test script that provided the code to implement the test case. Execution of the test script produced a test results file. There were other intermediate artifacts produced in the execution process, but these were the artifacts that required configuration control.

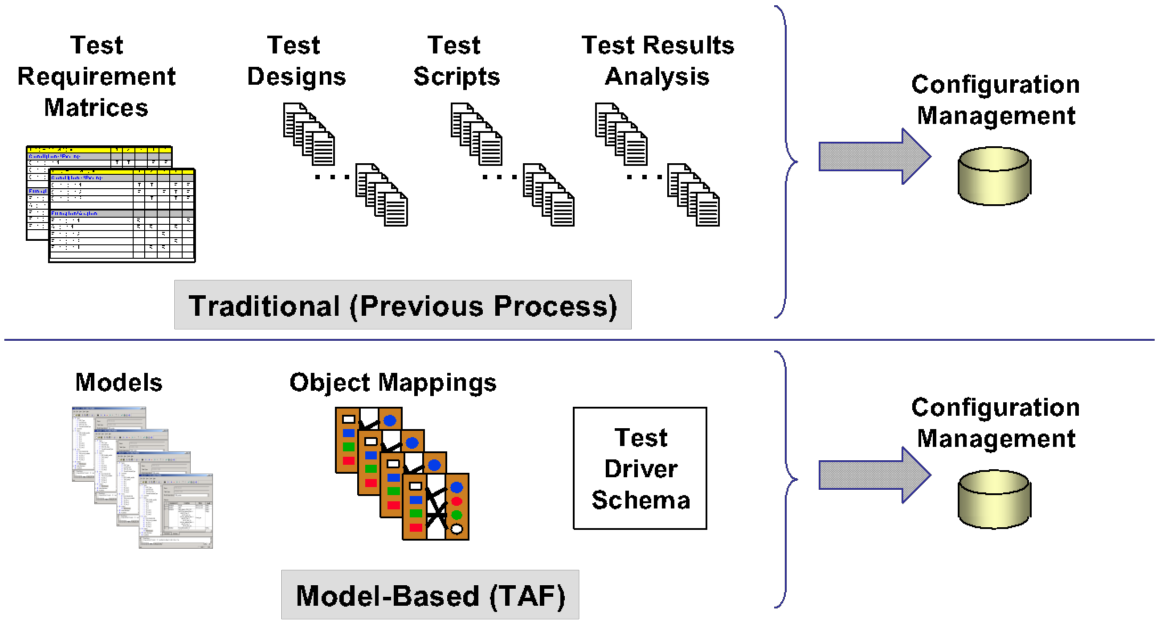


Figure 71. Test-Related Artifacts for Configuration Management

The model-based process involves the configuration control on the model (an XML file), which represents the requirements and has traceability links within the model back to the requirement specification. These links are traced forward to the test vectors automatically during the test vector generation process. The test design process is inherent in the model-based generation process. Test drivers that are generated from the object mappings and a test driver schema are not maintained under configuration control because they can be regenerated. The key benefit, as discussed in Section 14.3.6, is that detailed artifacts are produced by common elements from generators; for example, if a particular interface changes, only one object mapping needs to be changed, and all test scripts associated with that object mapping are regenerated. In contrast, with the traditional process, each test design and test script that references a particular input would need to be checked out of configuration control, modified, and then checked in after proper reviews were conducted.

14.4 KEY GUIDELINES

Most of the key guidelines provided in this section are listed in the summary provided in Section 15.1.

14.5 RESULTS

The results were significant. The company realized improvements in the way that technical specifications were developed. Early modeling helped identify requirement problems early, allowing changes to be made at minimal cost, as opposed to the older process where all changes would go through a formal change control board.

The improved design for testability reduced the effort by the designers and implementers that used early tests to reduce their unit testing effort. They were able to request that the test engineers add test constraints to the model requirements.

The test infrastructure group redesigned the simulation and test infrastructure services, reducing the variability in the way that the test services would be used and reducing the variation in the test execution.

The model-based test generation was more comprehensive than the manually generated tests cases. Also, it was efficient to have the tools generate test cases and tests scripts of hundreds of lines of code rather than producing and maintaining the test script code manually.

The models were easier to review, both from a requirements point –of view and a test completeness point –of view.

The bottom line is that the critical Phase III project was completed 9 weeks ahead of schedule, a new feat for this company. In the competitive marketplace, this company started on a different path to reduce cost and schedule to meet the high-assurance demands of systematic V&V.

This case study reflects the need for a highly successful company to *engineer its model-based testing*. There are many hidden benefits related not only to more effective testing but also to improved requirements and design. This case study reflects on this mature approach to model-based testing that has been implemented in a few member companies.

15. SUMMARY

These case studies provide evidence that model-based testing improves requirements, design, and tests, while increasing reliability and reducing cost and schedule. The most effective organizations have established an engineering-based approach to model-based testing.

This section summarizes some of the key guidelines in the report. These guidelines include both organizational and technical suggestions, with links where appropriate to specific case studies for details and examples.

15.1 SUMMARY OF KEY GUIDELINES

- ***Start with pilot projects to support organizational change***. Stakeholders need to see quickly demonstrated evidence within their organization to commit to use model-based testing on a scheduled deliverable. It is often good to select a pilot project from a recently completed project because the requirements are often well-understood, even if not well-documented. In addition, existing test cases and test results can provide a more objective basis for comparison with the model-based tests.

- ***Transition from a pilot project to a thread of an existing project***. Select a thread that is likely to change often or have features extend it. The most leverage and benefits come from reusing and evolving one or more related models and the associated test infrastructure. See Section 14 for details.

- ***Identify the right projects for transitioning from an existing process to a new process***. Select a project prior to the requirement phase so that modeling can start early and help improve the requirements, while providing sufficient time to collaborate with the design team to improve the interfaces to support testability.

- ***Start requirement modeling early***. Identifying requirement defects sooner reduces rework cost.

- Use interface-driven modeling to ensure the component under test has testable interfaces. Define the requirements for each component in terms of the known interfaces.

- ***Use a goal-oriented modeling approach***. Work backward by identifying each output at the component interfaces. Prioritize the ordering of the modeling for requirement threads to correspond with the expected development and/or integration of the component outputs associated with those requirement threads.

- ***Identify and model interfaces separately from behavioral requirements***. This approach maximizes reuse and ensures a single point of definition for each modeled interface. Include a model reference to interfaces for components that are related to the functional requirements they are modeling. This approach ensures that the model of an interface is defined in one place. If changes occur to the interface, only one model needs to be changed. See Section 14.3.6 for details.

- ***Capture requirement traceability links in the requirement models***. This approach provides important information to improve the review process. Tracing the requirements also helps in assessing the completeness of the model with respect to a requirement and related specification documents. Early modeling can identify incompletenesses in requirement documents that can be corrected early, providing better input to designers and implementers. See Sections 2.5.2.2 and 13.4.3 for examples and details.

- ***Ensure that requirement models capture negative cases as well as the positive cases of a requirement***. The negative case often can uncover problems such as the problem that was the likely cause of the MPL crash (see Sections 4.4) but also represent important safety or security cases as described in Section 12.4.3.

- ***Establish modeling practices***. Use practices such as naming conventions, terms that can be reused throughout the model, constants, and traceability links. See Section 10.4.1 for examples.

- ***Model continuously and in parallel with development***. This approach can reduce testing effort for designers and implementers and ensure that the design is testable. This also results in an evolving automated test suite that should be executed for every build (e.g., daily, bi-daily, or weekly) of the system. This approach also supports early identification of bugs and it makes it easier to understand and isolate the specific changes that introduced a defect into the system.

- ***Extend requirement-based test models***. Add test constraints to a model to support implementation-derived requirements identified by the designer or implementer to reduce the unit testing effort traditionally performed by implementers. See Section 14.3.5 for details.

- ***Leverage the expertise of test automation experts***. These experts often understand the most robust set of services for interfacing with the test environment as well as details related to initialization. See Section 14.3.6 for details.

- ***Develop common object mappings that correspond to modeled interfaces***. Ensure that the test driver schema isolates test environment specifics such as initialization and declaration that can be controlled by the test automation expert. See Section 14.3.6 for details.

- ***Develop and evolve one test driver schema per environment***. Coordinate effort through a lead test automation expert that leverages common logging, reporting, configuration management and measurement support. Ensure the test driver schema maps requirement identifiers to test scripts for more efficient test failure analysis. See Section 13.3.1 for details.

- ***Develop test driver schema that support output and state variable initialization.*** Ensure that the test driver schema properly initializes actual outputs to something other than the expected output value and properly initializes and propagates state data to expose potential failures. See Sections 4.4 for examples and details.

- ***Leverage internal monitoring capabilities of an application, such as a data recorder, to capture the actual outputs of a test***. For some types of applications that do not have internal monitoring, it is necessary to embed monitoring and logging functionality within the test driver so that, during test driver execution, the outcomes and state changes within the target environment are captured to support test results analysis. See Sections 8 and 9.4.5 for details.

- ***Model the dynamic generation of database content***. This modeling avoids the costly effort of developing and maintaining a "gold" database. See Section 12.4.1 for details.

- ***For more complex systems, analyze the interfaces, API dependencies, and requirement dependencies***. This analysis ensures proper design of models that should be associated with test driver object mappings. This can maximize the reuse of common interface models and object mapping definitions to reduce cost and maintenance and can reduce effort related to test sequencing. See Sections 13 and 14 for more details.

15.2 RESULTS AND BENEFITS

This set of case studies provides a short list of examples that summarize some of the benefits of model-based testing. Sections 15.2.1 through 15.2.4 provide a few other member company remarks with some perspective on the tangible as well as the intangible ROI associated with model-based test engineering.

15.2.1 Test Infrastructure Established During Pilot Project

One company stated there are many tangible benefits from model-based testing, but surprisingly, there are several intangible ROI benefits. At the end of the pilot demonstration, the process and the supporting test infrastructure were 80% to –90% complete and relatively stable to support all follow-on testing. In addition, the company identified several requirements for the testing infrastructure that could further automate the process or change the underlying process for the organization. For example: once an automated test suite exists, it can be run each time a build of the system occurs. This approach allows developers to identify bugs earlier in the development process and makes it easier to understand the specific changes that introduced a defect into the system rather than waiting weeks or months before manual testing is performed.

15.2.2 Completed Project Ahead of Schedule

The company that produces medical devices stated prior to the use of TAF that testing was more than 50% of their effort. Using model-based testing, they completed the development and verification 9 weeks ahead of schedule, a new feat for the company.

15.2.3 Significant Reduction in Retesting

The company producing medical devices stated that after a clinical trial of a product, they had to completely retest the entire product. They claimed that by using the model-based test suite that had been developed for the initial release, they were able to complete all their required testing five times faster than their existing script-based testing process.

A second member pointed out similar results. The key ROI gains are obtained with each addition regression testing session that occurs. Currently, the time required for regression testing is essentially the same as it is to test the first time. With this automated, model-based testing process, the time to perform regression testing is easily less than 50% of the original time and cost and can be as little as 10% of the original test time and cost.

15.2.4 TAF for Avionics and Aircraft Systems

Lockheed Martin has used the TAF for many years and has contributed significantly to the evolution and usage requirements. None of the case studies in this report reference the following applications, which come from this release citation:

> Some of the areas we have applied the TAF technology are in the Vehicle Systems Flight Control Laws, the Mission Systems Middleware, Digital Radio Controls, Auto Logistics AFB Basing and Flight Ops, Branch Health and Mode Determination software testing, and Reliability Enhancement and Re-engineering Program (RERP) Design and Test. These applications have been applied to various elements of multiple programs that include JSF/F-35, F2, and C5. Prior pilot applications performed on T-50 and F-16 showed applicability to legacy programs and would be beneficial in future upgrades to existing programs.
>
> Applications on future upgrades to existing program that have extensive tabular formatted requirements have been identified as highly adaptable to the TAF technology through the use of TTM and T-VEC.
>
> On one application related to Flight Control LAWS (Safety Critical software) it was determined that the application of TAF would significantly reduce the test efforts related to each release of the software. Typically there would be 6-12 releases for each version of this software with a total savings greater than six million dollars just in the test effort portion. Even more important would be the reduction in schedule time for the releases, which would result in greater dollar savings related to other personnel supporting the efforts.
>
> In one experience we discovered some critical errors (such as potential divide by zero) during the design effort that would not have been caught until the test phase of the development and in some cases may not have been detected until much later when these unique conditions were met. In another experience we discovered an error, early in the development cycle, with the code generation tool being used. Benefit analysis on software development shows that the cost of resolving these errors grows exponentially as they move through the development phases. The actual cost savings of these can only be imagined but definitely go into the millions for our type software.

15.3 CONCLUSION

These case studies provide evidence that model-based testing applies to many types of applications, such as embedded systems, language processing, client-server and web systems, distributed processing systems, command, control and monitoring, information processing business logic (IT systems), database, security, smart cards, and life-critical systems such as medical devices and avionics systems. Model-based has helped many organizations improve the requirements, driven improvements in the design of the target system and simulators, improved the test infrastructure with design for testability, guided the creation of modular requirement specifications, and demonstrated the cost benefits of modeling requirements early, with significant reduction in cost and schedule.

The TAF team helped organizations in the adoption process, including the structuring of a multiteam modeling and test infrastructure; recommended model-based review practices; created tailored training; and shown how to use model-based measurement for project management.

This page intentionally left blank

SCR REQUIREMENT MODELING

The modeling language for the SCR tool is similar in syntax to a simplified programming language. It has expressions for defining arithmetic computations and logic relationships, sometimes referred to as constraints. There are a few special language constructs that support the description of events. The semantics of an SCR model are declarative as opposed to imperative like a programming language.

This appendix discusses the rules for using the model, typically referred to as the method. As shown in Figure 72, inputs and outputs represent the system's interaction with the environment in which it operates. Most often, inputs and outputs directly represent component interfaces. Terms are intermediate computed values and provide the means to decompose the model hierarchically. Mode machines represent internal states as simple, finite state machines. Assertions are named conditional expressions that are referenced by name within the model.

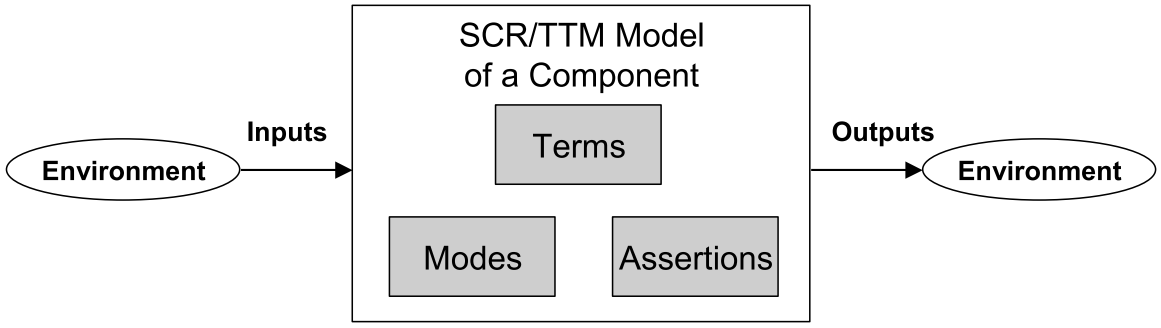


Figure 72. Top-Level Concept for Defining SCR Models

MONITORED AND CONTROLLED VARIABLES

A monitored variable corresponds with inputs of a function and an interface of the system. It represents a logic or environmental quantity that the system must monitor or use as input, and it must be mapped to an interface of the system or component under test.

A controlled variable corresponds with outputs of a function and an interface of the system. It represents a logic or environmental quantity that the system must set or control, and it must be mapped to an interface of the system or component under test.

REPRESENTING FUNCTIONAL VIEW

The functional view of a system is defined using behavioral elements to specify the set of relations between entities that represent the interfaces of the system. The behavioral aspects of the models define the required functionality of the component using tables to relate monitored variables (inputs) to controlled variables (outputs). 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

The SCR modeling approach permits Condition, Event, and Mode Tables to be combined. This allows complex relationships between monitored and controlled variables to be described in terms of simpler relationships that are modeled in Condition, Event, or Mode Tables. The concept of dependency relationships is supported using a mode class or a term variable, as shown in Figure 73. A *mode class* is a state machine, where related system states are called system modes, and the transitions of the state machine are characterized by events. A *term* is any function in input variables, modes, or other terms. A *condition* is a predicate characterizing a system state. An *event* occurs when any system entity changes value.

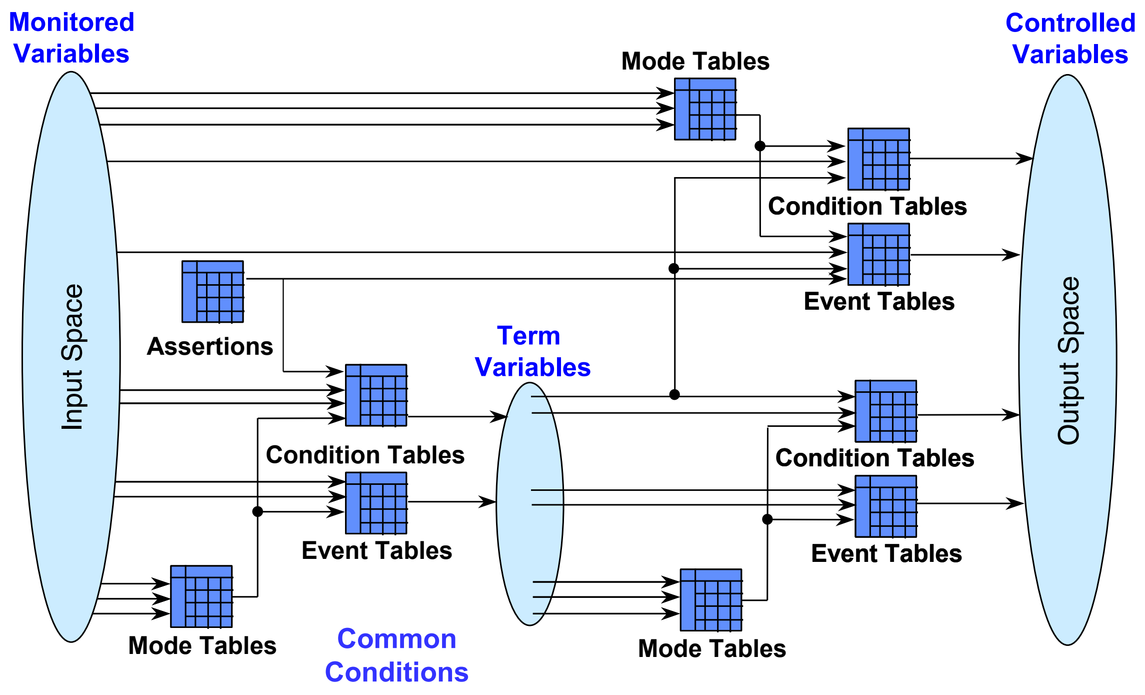


Figure 73. Conceptual SCR/TTM Model of Table Dependencies

CONDITIONS

A condition is a predicate (i.e., a statement that is true or false) about the values of monitored, controlled, term, or mode class variables. A condition is represented by a Boolean expression. Compound conditions are formed by using logical operators AND, OR, and NOT. For example, given conditions C, C1, and C2:

```
Compound conditions:              Is true when:
```

```
NOT C                               C is not true
C1 AND C2                           Both C1 and C2 are true
C1 OR C2                            C1 OR C2 OR both are true
```

The operations are listed in the descending order of precedence. Use parentheses to alter the evaluation order. By definition, each compound condition is also a condition.

EVENTS

An *event* occurs when a condition changes value. Hence, any condition has two kinds of events associated with it: those events that occur precisely when the condition changes from false to true and those that occur when the condition changes from true to false. An *event occurrence* is a moment in time when a condition's value changes. Each event occurrence is instantaneous (takes zero time) and atomic (all or none occurs).

Event expressions are used to represent the set of events associated with a particular condition. SCR uses the notations @T(C) and @F(C) to represent event expression denoting change in the state of a condition C. An event denoted by @T(C) occurs at any moment in time when the condition C transitions from false to true (Boolean expression evaluates to true). Similarly, @F(C) signifies any event of the condition C becoming false.

For example, consider the monitored variable mon_Push_Button, representing the state of a push button. At any moment in time, the button is in one of two possible states: pressed or released. The following event expressions denote the events corresponding to changes in the state of the button:

```
Event expression:              Represents:
@T(mon_Push_Button = pressed)        The event corresponding to a change 
in the 
                                      state of the button from released 
to pressed.
@F(mon_Push_Button = pressed)        The event corresponding to a change 
in the 
                                      state of the button from pressed 
to released.
```

The bulleted list provides an inductive definition of SCR events. In order to define such a variable it is enough to supply a name and an event expression. In contrast with other variables, an event variable does not have an initial value and is not considered to be a part of the system state. Also, an event variable name may optionally contain the symbol @ as a first character to distinguish from the other kinds of variables.

- A Boolean combination of event expressions is also an event expression.
- Basic events. Let c be a Boolean expression, and let f be an arbitrary expression such that neither c nor f includes occurrences of event variables. Then the following are basic event expressions:
  - @T(c) means that the value of c becomes TRUE in the current state whereas it was FALSE in the old state. @T(c) represents c' AND NOT 'c.

- @F(c) means that the value of c becomes FALSE in the current state, whereas it was TRUE in the old state. @T(c) represents NOT c’ AND ‘c.
- @CHANGED(c) means that the value of c changed from the old to the new state. @CHANGED(c) represents c’ != ‘c (where "!=" means ≠; in SCR/CoRE textual notation).

GUARDED EVENTS

Guarded events are basic events that are combined logically with a conditional expression. In order for guarded events to occur, some constraint must hold “WHEN” the event expression is satisfied. Some organizations have found that WHEN semantics are not sufficient for modeling their systems and have developed alternative guards referred to as WHERE and WHILE. This section introduces these alternatives. The following alternatives are supported in the scr2tvec translator through a command-line option, but they are not currently supported in the SCRtool:

- "WHEN" events. Let d be a Boolean expression, and let e be an event expression. Then, e WHEN d means that e occurred and that d was TRUE in the old state, where the event expression e is represented by @T(c). In general, e WHEN d represents e AND ‘d.
- "WHERE" events. Let d be a Boolean expression, and let e be an event expression. Then, e WHERE d means that e occurred and that d is TRUE in the new state, where the event expression e is represented by @T(c). In general, e WHERE d represents e AND d’.
- "WHILE" events. Let d be a Boolean expression, and let e be an event expression. Then e WHILE d means that e occurred and that d is TRUE in both the old and new states, where the event expression e is represented by @T(c). In general, e WHILE d represents e AND ‘d AND d’.

In order to define events, the state before the transition, also called the "old" state, and the state after transition, also called the "new" state must be specified. The transition between the old and the new states is done in several steps defined by the tables associated with the variables. The states between the old state and new state are called transient states. An important constraint associated with the state transitions states that for any state transition and for any variable, the variable is allowed to change its value once, at most, in the sequence of states including the old state, the new state, and all the transient states in between. This constraint is called the Single Change Constraint (SCC).

Given a system entity *var*, the value of *var* pertaining to the old state is explicitly designated as *’var*, whereas the value of *var* pertaining to the new state is explicitly designated as *var’*. Usually, when there is no explicit denotation with an apostrophe, *var* may designate either the value of *var* pertaining to the old state or the value of *var* pertaining to the new state, depending on context. For example, in WHEN (... *var* ...), *var* is treated as *’var*, whereas in WHERE (... *var* ...), *var* is treated as *var’*. The following list summarizes high-level state behavior of an SCR model (see Figure 1):

- A high-level state is an assignment of proper values to all the system entities.

- The states may be visible from outside (public) or be transient (private) states within a state transition between two public states.
- A high-level state transition is initiated by an external event represented by a modification of the current value of a unique monitored variable (One Input Assumption [OIA]).
- The initial event causes a ripple (cascade) of internal events where:
  - Every entity is modified at most once (due to SCC).
  - Monitored variables are not modified after the initial event (due to SCC).

TERMS

The definition of several controlled variable functions depends on the expression of monitored variables. Rewriting the same expression through the specification can be tedious and error-prone. To simplify the specification and to not such dependencies explicitly, SCR provides terms. A term is a name expression for one or more monitored variables. Each term has a value and a type of its constituent monitored variables and the operator applied.

Using terms, to abbreviate or replace lengthy expression with names. The notion of term in SCR is analogous to the concept of language macros (textual replacements) in some programming languages. The following list provides some of the most common reasons for defining terms:

- To shorten a complex or lengthy event expression used in one or more Event Tables or compound condition used in one or more Condition Tables. The use of properly defined terms reduces inconsistencies and improves the clarity of the model.
- To abstract a complex expression and hide its details. The modeler may not have finalized the details or may want to change them later.

This page intentionally left blank

CODE COVERAGE AND STRUCTURAL TESTING

Potential users of TAF and T-VEC often ask about support for code coverage. At the higher levels of safety and criticality, DO-178B requires evidence showing 100% structural code coverage from tests executed against the code. As shown in Figure 74, TAF/T-VEC provide model coverage; that is, from the model, the tools check to make sure that a test vector is produced for every translated thread of the model. If a test vector is not produced, the model has a defect, and the coverage report provides a link to that particular thread where the model defect is likely to exist. See Section 10.4.3) for further details.

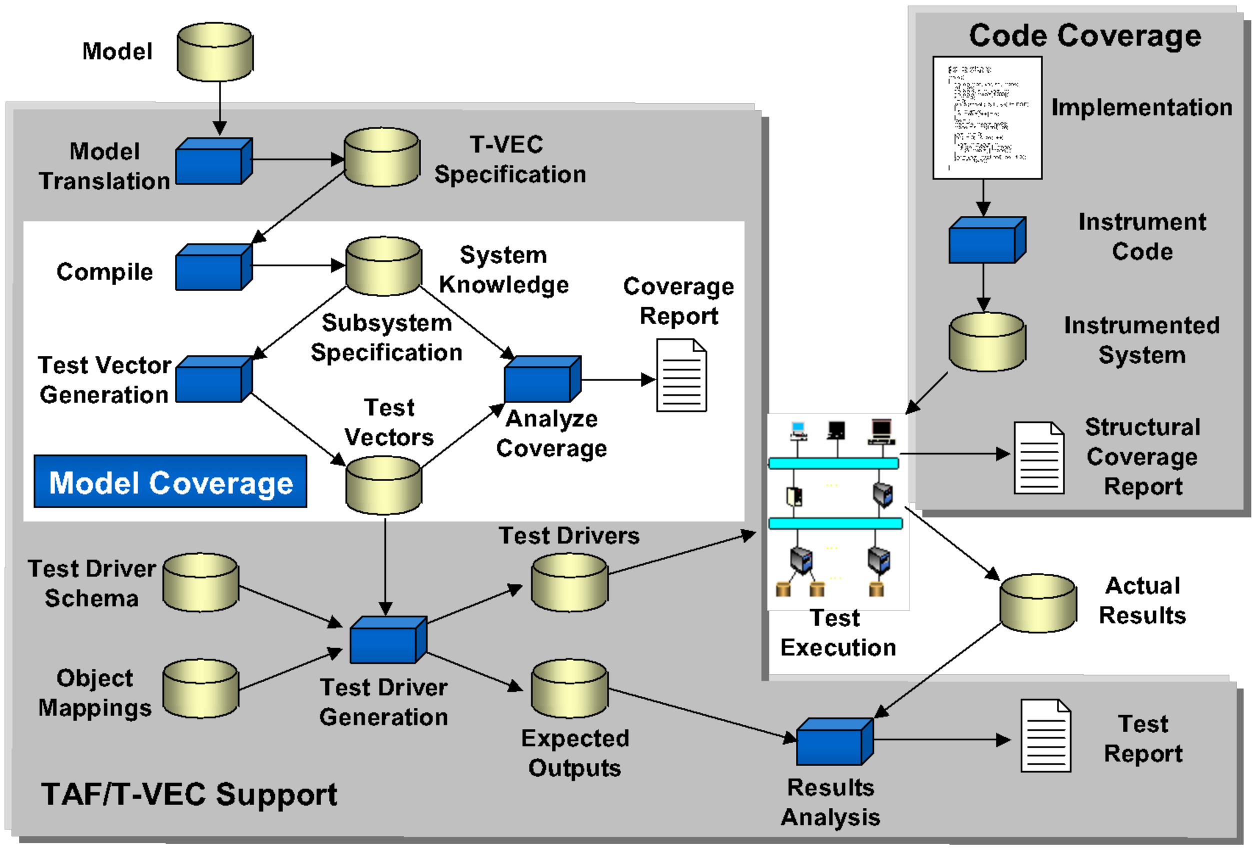


Figure 74. TAF Model Coverage Versus Code Coverage

The following definitions apply to structural testing and the associated code coverage:

- ***Condition.*** A condition is a leaf-level Boolean expression. It cannot be broken down into a simpler Boolean expression.

- ***Decision.*** A decision is a Boolean expression that controls the flow of the program, for instance, when it is used in an **if** or **while** statement. Decisions may be composed of a single condition or expressions that combine many conditions.

Structural testing criteria characterize the level of coverage of the code:

- ***Statement Coverage*****.** Every statement in the program has been executed at least once.
- ***Decision Coverage*****.** Every point of entry and exit in the program has been invoked at least once, and every decision in the program has taken all possible outcomes at least once.
- ***Condition/Decision 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 all possible outcomes at least once; and every decision in the program has taken all possible outcomes at least once.
- ***Modified Condition/Decision Coverage (MC/DC)*****.** 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.

There are various ways to assess code coverage, and there are some qualified tools that provide code coverage measurements. The typical process involves instrumenting the code that is produced manually or through code generation, executing the tests against the instrumented code, and assessing the code coverage. The TAF tools are integrated with different code coverage tools as reflected in Figure 74. One qualified tool is produced by LDRA as shown in Figure 75.

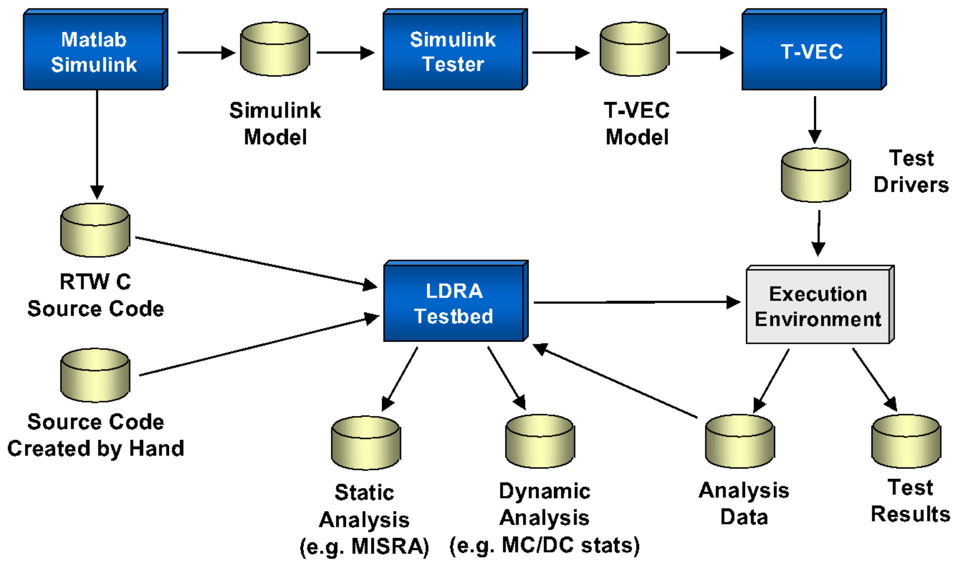


Figure 75. LDRA and TAF Integration

The TAF/T-VEC tools have translation options to create model threads that are capable of achieving MC/DC test coverage. Consider the model shown in Figure 76 and the associated code coverage information shown in Figure 77, which indicates that 12 out of the 12 paths were covered by the generated tests. If all tests pass, and there is complete code coverage, then there is a strong argument that the code fully satisfies the specified functionality of the model. The final step for testing the code is running the same tests through target code that will be the final certified code. If all tests pass, there is a strong argument that the code is suitable for certification.

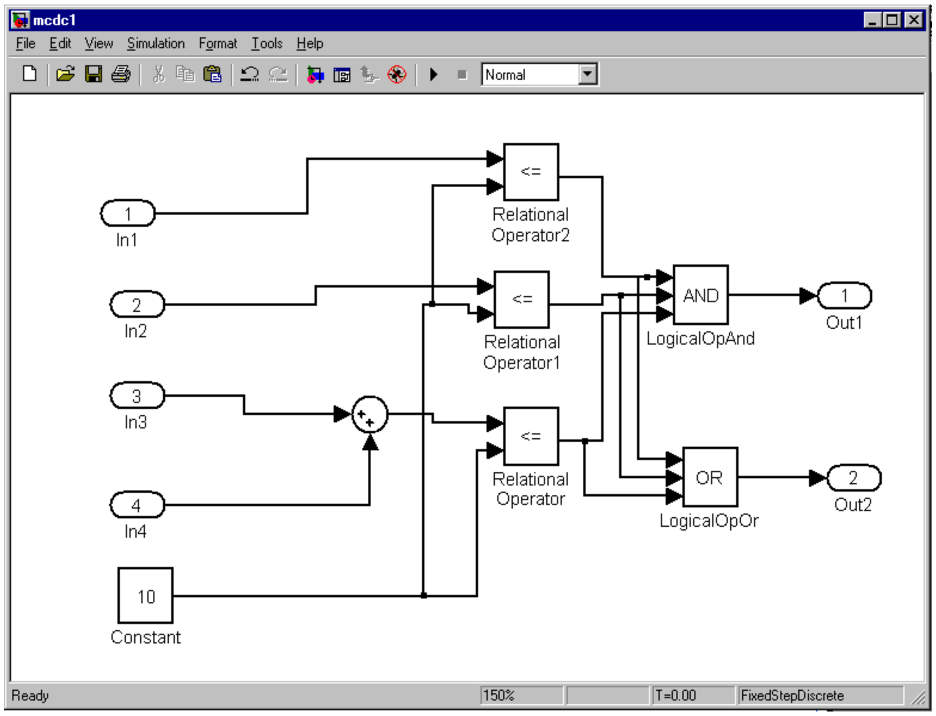
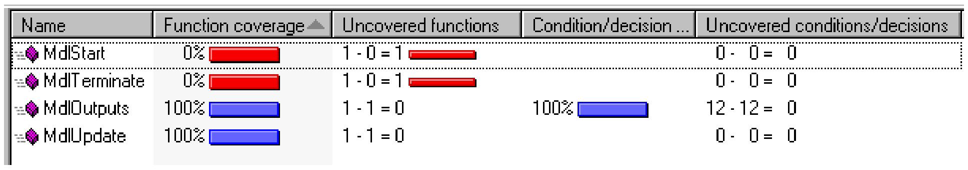
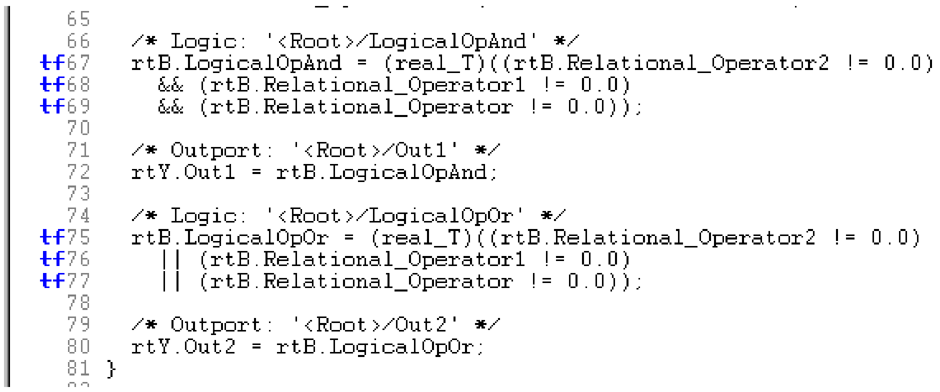


Figure 76. Simple Model for Code Coverage Example



Name	Function coverage	Uncovered functions	Condition/decision ...	Uncovered conditions/decisions
MdlStart	0% 🟥	1 - 0 = 1 🟥		0 - 0 = 0
MdlTerminate	0% 🟥	1 - 0 = 1 🟥		0 - 0 = 0
MdlOutputs	100% 🟦	1 - 1 = 0	100% 🟦	12 - 12 = 0
MdlUpdate	100% 🟦	1 - 1 = 0		0 - 0 = 0



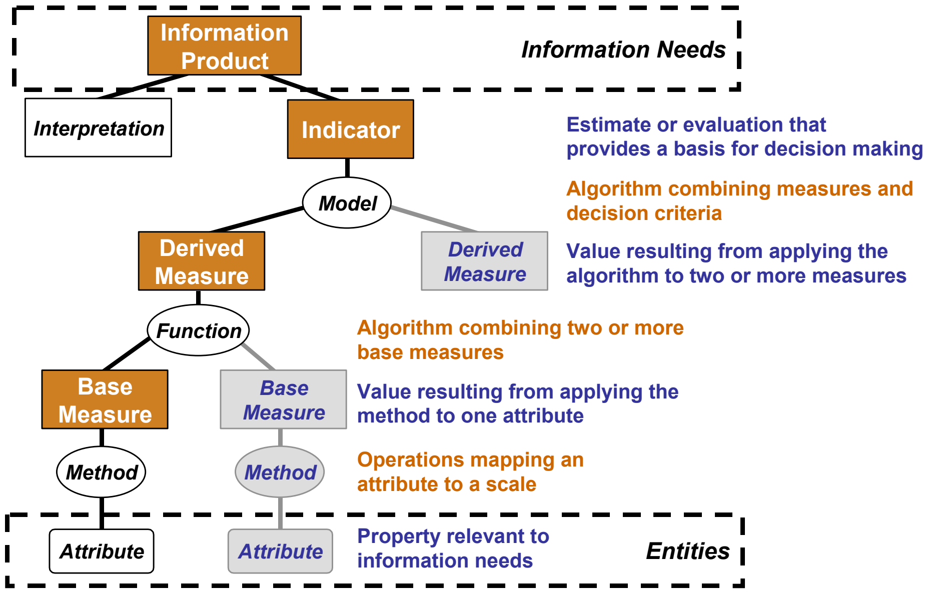
```

      /* Logic: '<Root>/LogicalOpAnd' */
tf    rtB.LogicalOpAnd = (real_T)((rtB.Relational_Operator2 != 0.0)
tf      && (rtB.Relational_Operator1 != 0.0)
tf      && (rtB.Relational_Operator != 0.0));

      /* Outport: '<Root>/Out1' */
      rtY.Out1 = rtB.LogicalOpAnd;

      /* Logic: '<Root>/LogicalOpOr' */
tf    rtB.LogicalOpOr = (real_T)((rtB.Relational_Operator2 != 0.0)
tf      || (rtB.Relational_Operator1 != 0.0)
tf      || (rtB.Relational_Operator != 0.0));

      /* Outport: '<Root>/Out2' */
      rtY.Out2 = rtB.LogicalOpOr;
    }
```

Screenshots from Bullseye Coverage

Figure 77. Coverage Analysis Screenshot

MEASUREMENT INFORMATION PRODUCT AND MEASUREMENT CONSTRUCT

The measurement and analysis process provides the mechanisms for identifying and addressing information needs of all types and levels. It addresses both the selection of appropriate measures to satisfy the information needs and the collection and analysis of the data. Information products make up the primary output of the measurement process. The generic structure of a measurement construct can be defined in terms of the information model, as shown in Figure 78. The information model shows how the attributes of specific software and systems entities are related to the information needs of the measurement user. This section works bottom-up from the information model of the measurement construct to describe the relationship of the TAF attributes, base measures, derived measures, and indicators to various information products supporting project measurement.


Information Product
Information Needs
Interpretation
Indicator
Estimate or evaluation that provides a basis for decision making
Model
Algorithm combining measures and decision criteria
Derived Measure
Derived Measure
Value resulting from applying the algorithm to two or more measures
Function
Algorithm combining two or more base measures
Base Measure
Base Measure
Value resulting from applying the method to one attribute
Method
Method
Operations mapping an attribute to a scale
Attribute
Attribute
Property relevant to information needs
Entities

Source: Adapted from ISO/IEC 15939, *Software Measurement Process Framework by Joe Seppy*

Figure 78. Measurement Construct

ATTRIBUTES

An attribute is a property or characteristic of a process or product that is the subject of measurement. Four attributes support measurement analysis: DCPs (requirement threads), object mappings, requirements, and variables.

BASE MEASURES

A base measure is a quantification of a single attribute obtained from some method or operation. There are several possible base measures that can be obtained through operations associated with the TAF attributes. Table 7 defines some key base measures.

Table 7. TAF Base Measures

Base Measure	**Explanation**
Number of DCPs	DCPs produced during a particular measurement period
Number of variables	Input and output variables produced during a particular measurement period that require a corresponding object mapping
Number of object mappings	Object mappings specified during a particular measurement period
Total number of DCPs	Sum of the DCPs measured from the start of the project
Total number of variables	Sum of input and output variables measured from the start of the project
Total number of object mappings	Sum of specified object mappings measured from the start of the project
Total number of requirements	Sum of total requirements being modeled using TAF for the project

To better understand the progress being made during project development, it is necessary to measure the DCPs, object mappings, and variables produced each week (these measures could be tracked daily, monthly, or yearly). Although it is idealistic to think that the total number of requirements will remain constant for a project, this seldom happens, so the total number of requirements for a project also should be monitored weekly. Fortunately, the total number of DCPs is generated as part of the status report for a project. Similarly, number of variables and object mappings produced each week can be recorded at the same time as the DCPs, etc.

DERIVED MEASURES

A derived measure combines two or more base measures using a mathematical function. For example, the *requirements per DCP* is a derived measure relating the number of requirements to modeled-derived DCPs. If a project manager can estimate the total number of requirements that must be modeled, it is possible to predict the total number of DCPs for the project. If the rate of DCP production per week were known, the scheduled end date could be predicted. This report assumes that the measurement period is weekly. Table 8 identifies some derived measures.

Table 8. TAF Derived Measures

Derived Measure	Explanation
DCP rate	The average number of DCPs produced per week
DCP rate (current week)	The average number of DCPs produced per week at some week, where current week is a week number starting from the beginning of the project
DCP density	The number of DCPs per requirement
Object mapping rate	The average number of object mappings specified per week
Object mapping rate (current week)	The average number of object mappings specified per week at some week
Variables per DCP	The number of input and output variables per DCP

These types of derived measures can be associated with an individual, project, project team, product, product family, business unit, or an entire corporation. Project teams as well as project team members will have different DCP rates and object mapping rates. These rates can vary based on the modeling skills of the team members, and can be affected by requirement rework caused by poorly documented or unknown requirements or poorly defined interfaces.

INDICATORS

An indicator is a base or derived measure or a combination of such measures that are associated with decision criteria by means of a mathematical or heuristic model. Indicators often are presented in graphic or chart form. Consider the following example that illustrates the use of the measurement construct. Assume that project decision makers want to compare the DCP density of some current project to previously captured data because it is believed that a DCP density in a certain range provides optimal requirement-to-test traceability. Assume the following:

- Base measures for DCPs and requirements have been collected.
- The number of requirement headers is the base measure for number of requirements, where a requirement header is associated with a body of requirement text.
- The data collected from prior projects estimates the number of DCPs per requirement.

Starting from the bottom of Figure 79, the two attributes are DCPs and requirements. Assume some sample project developed 223 DCPs in models for 21 requirements. Dividing the number of DCPs by the number of requirements produces a derived measure of 10.6 DCPs per requirement (i.e., DCP density). Compare this derived data to historical data, where the density was 16.3 DCPs per requirement. A possible interpretation of this information product is that the requirement traceability accuracy for the current project is better than the organizational average. If the current DCP density has a variance greater than 10 from the organizational average, then it may be necessary for the requirement engineers to attend a training class on techniques for developing better requirements.

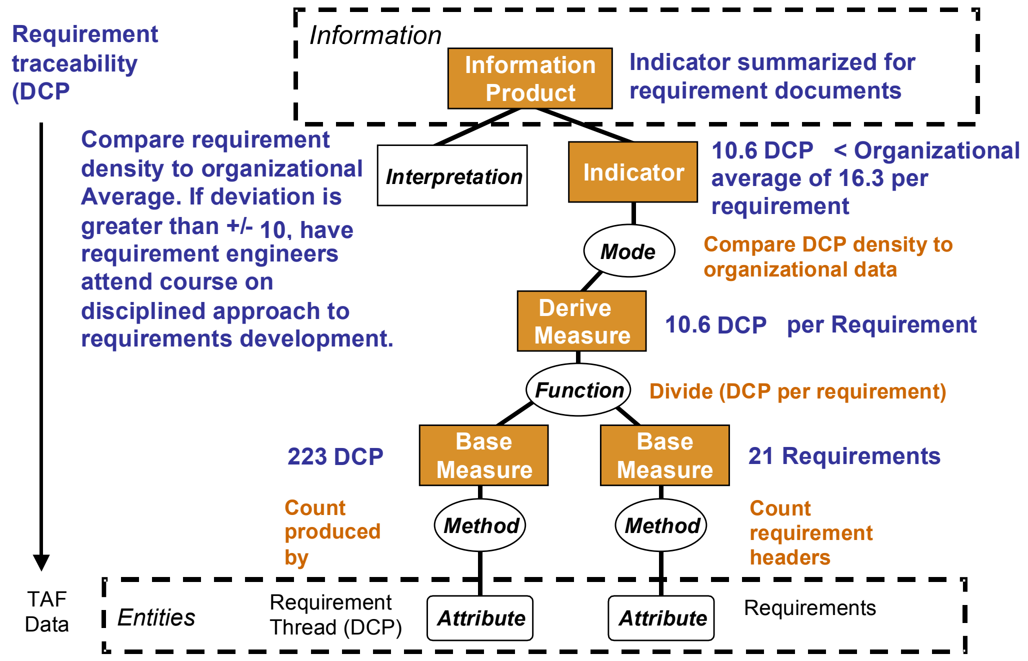

Requirement traceability (DCP
Information
Information Product
Indicator summarized for requirement documents
Compare requirement density to organizational Average. If deviation is greater than +/- 10, have requirement engineers attend course on disciplined approach to requirements development.
Interpretation
Indicator
10.6 DCP < Organizational average of 16.3 per requirement
Mode
Compare DCP density to organizational data
Derive Measure
10.6 DCP per Requirement
Function
Divide (DCP per requirement)
223 DCP
Base Measure
Base Measure
21 Requirements
Count produced by
Method
Method
Count requirement headers
TAF Data
Entities
Requirement Thread (DCP)
Attribute
Attribute
Requirements

Figure 79. Measurement Construct Example

LIST OF ABBREVIATIONS AND ACRONYMS

Accreditation Process	
ACR	Access Control Rules
AID	Application Identifiers
APDU	Application Protocol Data Units
API	application programming interface
BNF	Backus-Naur Form
BSI	Basic Service Interface
CCC	Card Capability Container
CICS	Customer Information Control System
CoRE	Consortium Requirements Engineering
CP	Command Processor
CST	Common Criteria Security Target
DBA	database administrator
DBMS	database management system
DCP	Domain Convergence Path
DP	Data Processor
DTD	Document Type Declaration
ECI	Environment Control Interface
FAA	Federal Aviation Administration
FDA	Food and Drug Administration
FGS	Flight Guidance System
FIPS	Federal Information Processing Standard
FMS	Flight Management System
FTP	File Transfer Protocol
GSC-IS	Government Smart Card Interoperability Specification
GUI	graphical user interface

HTML	HyperText Markup Language
ID	identifier
IEC	International Electrotechnical Commission
IEEE	Institute for Electrical and Electronics Engineers, Inc.
IP	Interface Processor
IPP	Internet Printing Protocol
ISO	International Organization for Standardization
IT	information technology
JCL	Job Control Language
JDBC	Java Database Connectivity
LDRA	Liverpool Data Research Associates
MC/DC	Modified Condition/Decision Coverage
MPL	Mars Polar Lander
NIST	National Institute of Standards and Technology
ODBC	Open Database Connectivity
OIA	One Input Assumption
OS	operating system
PKI	public key infrastructure
RERP	Reliability Enhancement and Re-engineering Program
ROI	return on investment
SCC	Single Change Constraint
SCR	Software Cost Reduction
SFR	Security Functional Requirement
SPS	Strategic Problem Solving
SPS	Strategic Problem Solving
SQL	Structured Query Language
SRS	System Requirement Specification
TAF	Test Automation Framework
TCAS	Traffic and Collision Avoidance System
TDM	Touchdown Monitor
TOE	Target of Evaluation
TSF	TOE Security Functionality
TTM	T-VEC Tabular Modeler

UK MoD	United Kindom Ministry of Defence
V&V	Validation and Verification
VCEI	Virtual Card Edge Interface
VM	virtual machine
XML	eXtensible Markup Language
XPIF	Printing Instructions Format Specification

This page intentionally left blank

REFERENCES

[Alspaugh 1992]	Alspaugh, T.A., S.R. Faulk, K.H. Britton, R.A. Parker, D.L. Parnas, and J.E. Shore. *Software Requirements for the A-7E Aircraft*, Tech. Rep. NRL/FR/5546-92-9194. Washington, DC: Naval Research Lab, 1992.
	Blackburn, M.R., R.D. Busser, A.M. Nauman, R. Knickerbocker, and R. Kasuda. *Mars Polar Lander Fault Identification Using Model-based Testing*. Lockheed Martin Joint Symposium, Radisson Universal, Orlando, 2001.
[Blackburn 2003]	Blackburn, Mark, Robert Busser, and Aaron Nauman. “Understanding the Generations of Test Automation.” Presented at STAREAST 2003, Orlando, Florida, May 12-16. http://www.software.org/pub/externalpapers/understanding_generations_of_test_automation.pdf
[Blackburn 2004]	Blackburn, M.R., A. Nauman. *Strategies for Web and GUI Testing,* SPC-2004014-MC, version 1.0. Herndon, Virginia: Software Producitivity Consortium, 2004.
[Boden 2004]	Boden, L., and R. D. Busser. “Adding Natural Relationships To Simulink Models To Improve Automated Model-Based Testing.” Digital Avionics Systems Conference, Salt Lake City, Utah, October 2004.
[Boehm 1984]	Boehm, B.W. “Verifying and Validating Software Requirements and Design Specifications.” *IEEE Software*. (January 1984).
[Busser 2001]	Busser, R.D., M.R. Blackburn, and A.M. Nauman. “Automated Model Analysis and Test Generation for Flight Guidance Mode Logic.” Digital Avionics System Conference, 2001.
[Consortium 1997]	Software Productivity Consortium. *Test Automation Framework*, SPC-97055-MC, version 01.01.00. Herndon, Virginia: Software Productivity Consortium, 1997.
[Consortium 1998]	Software Productivity Consortium. *Specification Transformation to Support Automated Testing*, SPC-97036-MC, version 03.00.02. Herndon, Virginia: Software Productivity Consortium, 1998.

[Consortium 2000]	Busser, R. D., M. R. Blackburn, and A. M. Nauman. *Rockwell Pilot Project*, SPC-2000045-MC, version 01.00.00. Herndon, Virginia: Software Productivity Consortium, 2000.
	Software Productivity Consortium. *Rockwell Pilot Project Technical Note*, SPC-2000045-MC, Version 01.00.00. Herndon, Virginia: Software Productivity Consortium, 2000.
	Software Productivity Consortium. *Applying the Test Automation Framework With Use Cases and the Unified Modeling Language*, SPC-2002048-MC, Version 01.00.00. Herndon, Virginia: Software Productivity Consortium, 2002.
[Consortium 2003a]	Software Productivity Consortium. *Model-Based Development and Automated Testing*, SPC-98070-MC, version 02.05.00. Herndon, Virginia: Software Productivity Consortium, 2003.
[Consortium 2003b]	Software Productivity Consortium. *Testing Complex Systems*, SPC-2003079-MC, version 01.00.00. Herndon, Virginia: Software Productivity Consortium, 2003.
[Consortium 2003c]	Software Productivity Consortium. *Guidelines for Software Tool Qualification*, SPC-2003064-MC, version 01.00. Herndon, Virginia: Software Productivity Consortium, 2003.
[Consortium 2003d]	Software Productivity Consortium. *Guidelines for Using Test Automation Framework Measures*, SPC-2003056-MC, version 01.00. Herndon, Virginia: Software Productivity Consortium, 2003.
[Consortium 2003e]	Software Productivity Consortium. *Test Automation Framework for T-VEC and Simulink*, SPC-2003048-MC, version 01.00 Herndon, Virginia: Software Productivity Consortium, 2003.
[Consortium 2004a]	Software Productivity Consortium. *Guidance for Achieving Mission Assurance in Software-Intensive Systems*, SPC-2004041-MC, version 01.00. Herndon, Virginia: Software Productivity Consortium, 2004.
[Consortium 2004b]	Software Productivity Consortium. *Model-Based Verification and Validation for Security Requirements of Systems*, SPC-2004034-MC, version 01.00. Herndon, Virginia: Software Productivity Consortium, 2004.

[Consortium 2004c] Software Productivity Consortium. *Requirement-Based Verification Sign-Off for Subcontract Integration Compliance*, SPC-2004022-MC, version 01.00. Herndon, Virginia: Software Productivity Consortium, 2004.

[Consortium 2004d] Software Productivity Consortium. *Automatic Code Generation: State of the Practice*, SPC-2004010-MC, version 1.0. Herndon, Virginia: Software Productivity Consortium, 2004.

[Consortium 2004e] Software Productivity Consortium. *Strategies for Web and GUI Testing*, SPC-2004014-MC, version 1.0. Herndon, Virginia: Software Productivity Consortium, 2004.

[Faulk 1993] Faulk, S.R., L. Finneran, J. Kirby, A. Moini, *Consortium Requirements Engineering Guidebook*, SPC-92060-CMC. Herndon, Virginia: Software Productivity Consortium, 1993.

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

[ISO 1995a] International Organization for Standardization. *Interindustry Commands for a Cryptographic Toolbox.* ISO/IEC 7816-8 (E). Geneva, Switzerland: International Organization for Standardization, 1995.

[ISO 1995b] International Organization for Standardization. *Interindustry Commands for Interchange*. ISO/IEC 7816-4 (E). Geneva, Switzerland: International Organization for Standardization, 1995.

International Standards Organization. *Common Criteria for Information Technology Security Evaluation.* Version 2.1, ISO/IEC 15408, CCIB-99-031, CCIB-99-032, CCIB-99-033. Geneva, Switzerland: ISO, August 1999.

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

[Offutt 1999] Offutt, A.J., Generating Test Data From Requirements/Specifications: Phase III Final Report, George Mason University, November 24, 1999.

[Oracle 2000] Oracle Corporation. *Oracle8 Security Target,* Release 8.0.5, Security Evaluations. Redwood Shores, California: Oracle Corporation, April 2000.

[Pettichord 2002]	Pettichord B. "Design for Testability ." Presented at the Pacific Northwest Software Quality Conference (PNSQC), October 2002. http://www.io.com/~wazmo/papers/design_for_testability_PNSQC.pdf
	DO-178B/ED-12B - *Software Considerations in Airborne Systems and Equipment Certification*, Radio Technical Corporation for Aeronautics Special Committee 167 (RTCA) December, 1992.
[Safford 2000]	Safford, Ed L. Test Automation Framework, State-based and Signal Flow Examples. In *Proceedings, 12th Annual Software Technology Conference*. Salt Lake City, Utah, April 30-May 5, 2000.
[SSCI 2005]	Systems and Software Consortium, Inc. *Objective Measures for V&V and Software Reliability*, SPC-2004010-MC, version 01.00. Herndon, Virginia: Systems and Software Consortium, Inc., 2005.
[Tsai 1990]	Tsai, W. T., D. Volovik, and T. F. Keefe. “Automated Test Case Generation for Programs Specified by Relational Algebra Queries”, *IEEE Transactions on Software Engineering* 16(3):316-324, March 1990.
[White 1980]	White, L. J., and E. I. Cohen. “A Domain Strategy for Computer Program Testing.” *IEEE Transactions on Software Engineering* Vol. SE6(3):247-257, May 1980.