BIO

PAPER

W8

November 6, 2002
2:15 PM

INTERFACE-DRIVEN MODEL-BASED TEST AUTOMATION

Mark Blackburn
Robert Busser
Aaron Nauman
Software Productivity Consortium

International Conference On
Software Testing Analysis & Review
November 4-8, 2002
Anaheim, CA USA

Mark R. Blackburn, Ph.D.

Dr. Blackburn is a Software Productivity Consortium Fellow and co-inventor of the T-VEC system. He has twenty years of software systems engineering experience in development, management and applied research of process, methods and tools. He is currently involved in consulting, strategic planning, proposal and business development, as well as developing and applying methods for model-based approaches to support requirement defect removal and test automation. He has also been involved in applied research and technology demonstrations in requirement and design specification, formal methods, and formal verification, object technology, web-based knowledge engineering, domain engineering, and reverse engineering. He earned a BS in Mathematics from Arizona State, MS in Mathematics from Florida Atlantic University, and a Ph.D. in Information Technology from George Mason University.

Robert D. Busser

Mr. Busser is a Principal Member of Technical Staff at the Software Productivity Consortium and co-inventor of the T-VEC system. He has over twenty years of software systems engineering experience in development, and management in the area of advanced software engineering, and expertise in software engineering processes, methods and tools. He is the chief architect of the T-VEC system. He has extensive experience in requirement and design methods, real-time systems, model-based development and test generation tools, model analysis, and verification. He has extensive knowledge about model transformation systems, theorem prover and constraint solving systems. In addition, he has extensive avionics engineering experience and has been involved in several FAA certifications. He has experience applying this knowledge in the development of highly-reliable software systems and the development of state of the art requirements-based software modeling and testing technologies. Mr. Busser has a B.S. in Electrical and Electronics Engineering from Ohio University.

Aaron M. Nauman

Mr. Nauman has a wide range of systems and applications development experience in both real-time (telecommunications) and information systems domains. He is currently involved in the development of model transformation, and software verification through specification-based automated testing. His experience includes all aspects of product development from requirements analysis through test implementation. Additionally, he has experience in object-oriented technologies, distributed and client/server systems, web-based and components-based software and systems integration. He is a representative on the OMG UML Action Semantics working group. Mr. Nauman graduated Summa Cum Laude from North Carolina State University with a B.S. in Computer Science.

Interface-driven Model-based Test Automation

Mark Blackburn, Robert Busser, Aaron Nauman
(Software Productivity Consortium)

This paper describes an interface-driven approach that combines requirement modeling to support automated test case and test driver generation. It focuses on how test engineers can develop more reusable models by clarifying textual requirements as models in terms of component or system interfaces. The focus of interface-driven modeling has been recognized as an improvement over the process of requirement-driven model-based test automation. The insights and recommendations for applying this approach were identified by model-based testing users when they began to scale their model-based testing efforts on larger projects.

Keywords: Test Automation Technology and Experience, Interface-driven Model-Based Test Automation, Test Driver Generation, Requirement-based Testing

1 Introduction

Model-based test automation has helped reduce cost, provide early identification of requirement defects, and improve test coverage [RR00; KSSB01; BBNKK01; BBN01d; Sta00; Sta01]. Industry use of model-based test automation has provided insights into practical methods that use interface-driven analysis with requirement modeling to support automated test generation. The interface analysis provides key information that results in **test driver mappings** that specify the relationships between model variables and the interfaces of the system under test. Recommendations are provided for performing the modeling of textual requirements in conjunction with interface analysis to support reuse of models and their associated test driver mappings. The insights are useful for understanding how to scale models and the associated test driver mappings to support industry-sized verification projects, while supporting organizational integration that helps leverage key resources.

1.1 Background

We have been applying the model-based test automation method referred to as the Test Automation Framework (TAF) since 1996. TAF integrates various government and commercially available model development and test generation tools to support defect prevention and automated testing of systems and software. TAF supports modeling methods that focus on representing requirements, like the Software Cost Reduction method, as well as methods that focus on representing design-level information, like Unified Modeling Language (UML)-based tools or Mathwork's Simulink, which supports control system modeling for automotive and aircraft systems. These modeling perspectives are discussed in Section 2.3.1. Through the use of model translation, requirement-based or design-based models are converted into test specifications. T-VEC is the test generation component of TAF that uses the test specification to produce tests. T-VEC supports test vector generation, test driver generation, requirement test coverage analysis, and test results checking and reporting. **Test vectors** include inputs as well as the expected outputs with requirement-to-test traceability information. The test driver mappings and the test vectors are inputs to the test driver generator that produces test drivers that are executed against the implemented system during test execution.

TAF has been applied to applications in various domains including critical applications for aerospace (Mars Polar Lander) [BBNKK01], medical devices, flight navigation, guidance, autopilots, display systems, flight management and control laws, engine controls, and airborne traffic and collision avoidance. TAF has also been applied to non-critical applications like databases, client-server, web-based, automotive, and telecommunication applications. The related test driver generation has been developed for many languages (e.g., C, C++, Java, Ada, Perl, PL/I, SQL, etc.) as well as proprietary languages, COTS test injection products (e.g., DynaComm®, WinRunner®) and test environments. Most users of the approach have reduced their verification/test effort by 50 percent [KSSB01, Saf00].

1.2 Contributions

This paper provides recommendations for combining interface analysis and requirement modeling. These recommendations for defining interfaces that provide better support for testability are valid for all forms of testing.

1.3 Organization of Paper

Section 2 provides an overview of the method and tools, while providing concept definitions, guidance on interface definitions and analysis, and organizational roles and best practices. Section 3 discusses requirements modeling using the Mars Polar Lander example. Section 4 discusses the organizational impacts of using interface information to support test driver mappings and associated support required for test driver generation. Section 5 summarizes the benefits of interface-driven model-based testing.

1.4 Related Work

There are papers that describe requirement-modeling [HJL96; PM91; Sch90], and others with examples that support automated test generation [BBN01a; BBN01b; BBN01c; BBNC01, BBNKK01]. Asisi provides a historical perspective on test vector generation and describes some of the leading commercial tools [Asi02]. Pretschner and Lotzbeyer briefly discuss Extreme Modeling that includes model-based test generation [PL01], which is similar to uses of TAF as discussed in Section 2.4. There are various approaches to model-based testing and Robinson hosts a website that provides useful links to authors, tools and papers [Rob00].

2 Method and Tool Overview

The TAF support, as shown in Figure 1, involves three main roles of development, including Requirement Engineer, Design/Implementation Engineer, and Test Engineer. A requirements engineer performs requirement analysis and typically documents the requirements in text. A designer/implementer develops the technical solution, which includes system/software architecture, design, components and interfaces, and implementation. Interfaces are typically documented in an application programming interface (API) or other interface documents. Test engineers clarify the requirements in the form of a **verification model**, which specifies behavioral requirements in terms of the interfaces for the system under test. This is in contrast to a “pure” requirement model, which specifies the requirements in terms of *logical entities* representing the environment of the system under test [PM91; Sch90; HJL96]. Verification modeling from the interfaces is analogous to the way a test engineer develops tests in terms of the specific interfaces of the system under test. TAF translators convert verification models into a form where the T-

VEC system generates test vectors and test drivers, with requirement-to-test traceability information that allows failures to be traced backwards to the requirement.

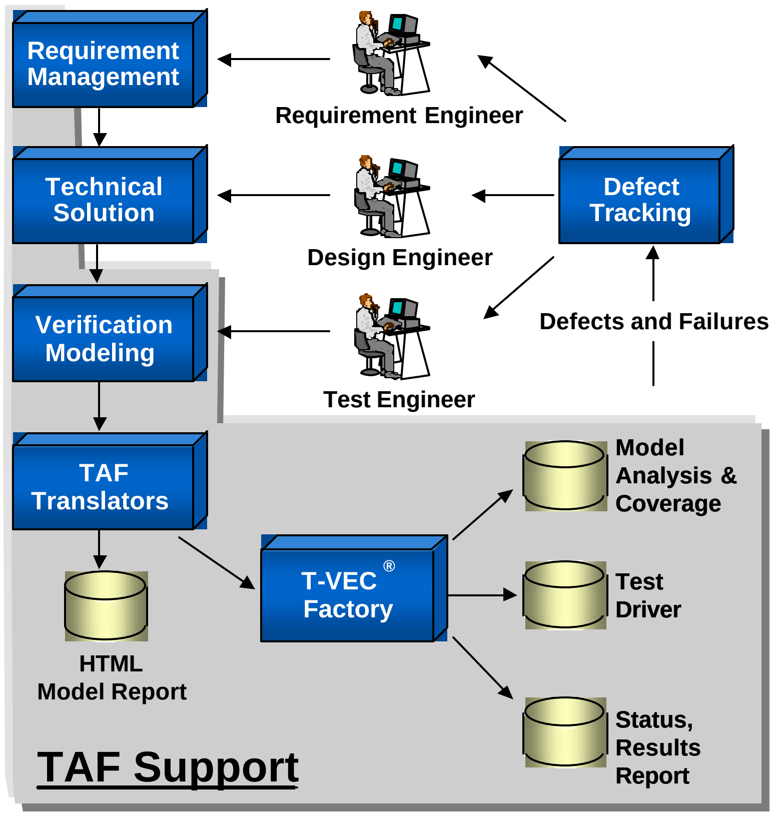


Figure 1. Test Automation Framework Life Cycle Automation

2.1 Verification Modeling Process

Figure 2 provides a detailed perspective of the verification modeling process flow. A test engineer is supplied with various inputs. Although it is common to start the process with poorly defined requirements, inputs to the process can include requirement specifications, user documentation, interface control documents, application program interface (API) documents, previous designs, and old test scripts. A verification model is composed of a model and one or more test driver mappings. A test driver consists of object mappings and a schema (pattern). Object mappings relate the model objects to the interfaces of the system under test. The schema defines the algorithmic pattern to carry out the execution of the test cases.

Models are typically developed incrementally. The models are translated and T-VEC generates test vectors. T-VEC also detects untestable requirements (i.e., requirements with contradictions). The generation of test vectors and defect detection does not use the test driver information. Test drivers are produced from the test vectors using the test driver mappings and schema information. Detail is provided in Section 4.

2.2 Why Tabular Models?

Table-based requirement modeling like the SCR method has been very effective and relatively easy to learn for test engineers [KSSB01]. Although design engineers commonly develop models based on state machines or other notations like the Unified Modeling Language (UML), users and project leaders observed that test engineers find it easier to develop requirements for test in the form of tables (See [BBN01a] for details). The modeling notations supported by tools for the SCR method have well-defined syntax and semantics allowing for a precise and analyzable definition of the required behavior. This paper provides an example of a functional tabular model for a small portion of the Mars Polar Lander requirements in Section 3.

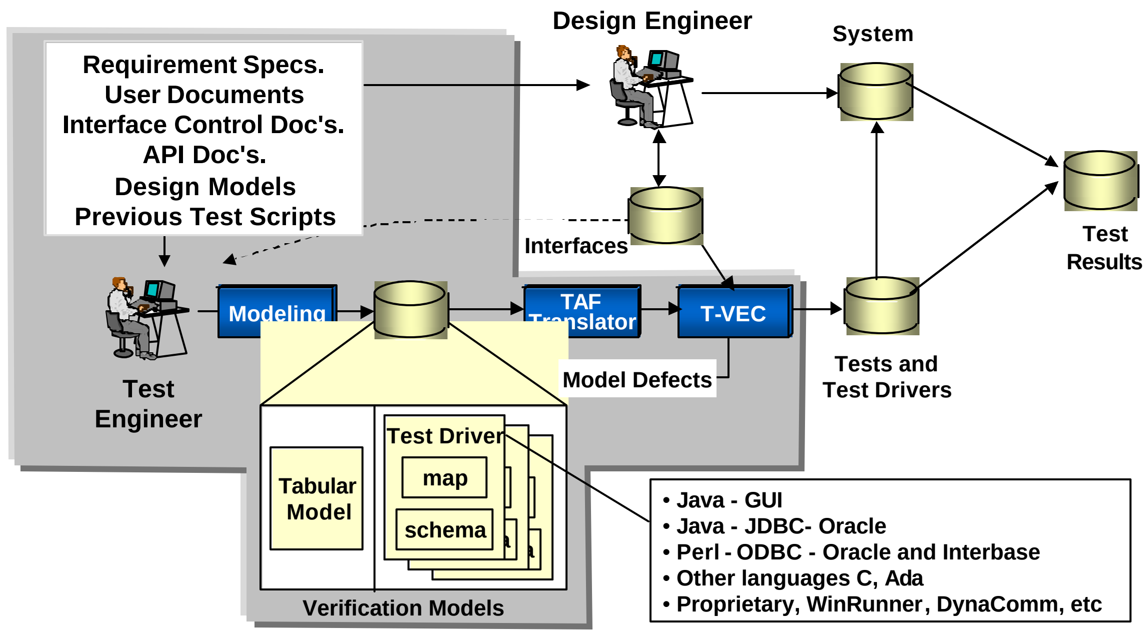


Figure 2. Verification Model Details

2.3 Why Interface-Driven Modeling?

It may seem appropriate to first develop models from the requirements, but when developing models for the purpose of testing, the models should be developed in conjunction with analysis of the interfaces to the component or system under test. Modeling the behavioral requirements is usually straightforward and easier to evolve once the interfaces and operations are understood because the behavioral requirements, usually defined in text, must be modeled in terms of variables that represent objects accessible through interfaces.

2.3.1 Modeling Perspectives

Models are described using specification languages, usually supported through graphical modeling environments. Specification languages provide abstract descriptions of system and software requirement and design information. Cooke et al. developed a scheme that classified specification language characteristics [CGDDTK96]. Independent of any specification language, Figure 3 illustrates three categories of specifications based on the purpose of the specification. Cooke et al. indicates that most specification languages usually are based on a hybrid approach that integrates different classes of specifications.

Requirement specifications define the boundaries between the environment and the system and, as a result, impose constraints on the system. Functional specifications define behavior in terms of the interfaces between components, and design specifies the component itself. A specification may include behavioral, structural, and qualitative properties. Behavioral properties define the relationships between inputs and outputs of the system [Sim69]; structural properties provide the basis for the composition of the system components; and qualitative requirements [YZCG84] define nonfunctional requirements. Often, languages support certain elements of requirement and functional specifications and are termed functional requirements, as opposed to nonfunctional requirements [Rom85].

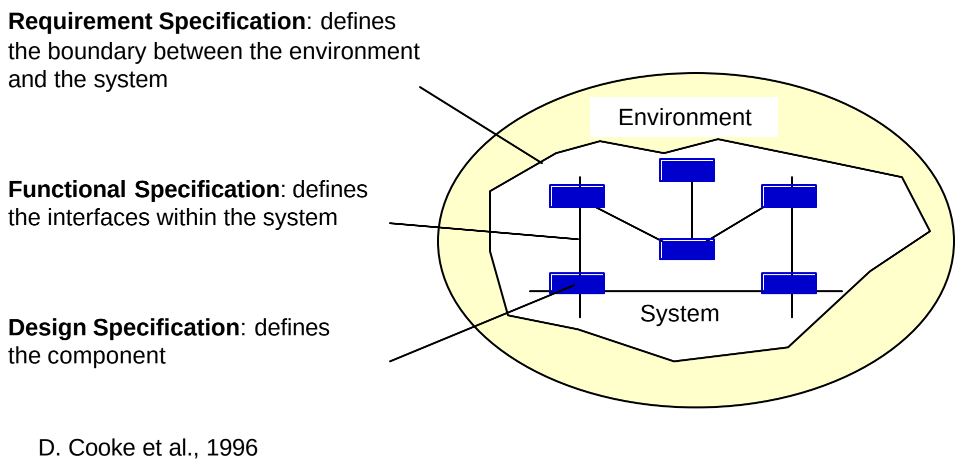

Requirement Specification: defines the boundary between the environment and the system
Functional Specification: defines the interfaces within the system
Design Specification: defines the component
Environment
System
D. Cooke et al., 1996

Figure 3. Specification Purposes

A verification model, in the context of this paper, is best classified as a functional specification. The requirements are defined in terms of the interfaces of the components. The term interface is used loosely in this paper. An **interface** is a component's inputs and outputs, along with the mechanism to set inputs, including state and history information, and retrieve the resulting outputs. Some components or systems may require sequences of function calls to initialize a component or system, as well as additional calls to place the system in a particular state prior to setting the inputs for testing.

2.3.2 Interface Accessibility

It is best to understand the interfaces of the system under test prior to modeling the behavioral requirements to ensure that the interfaces for the resulting test driver map to actual inputs or outputs of the system under test. If the interfaces are not formalized or completely understood, requirement models can be developed, but associated object mappings required to support test driver generation must be completed after the interfaces have been formalized. This can make the object mapping process more complex, because the model entities may not map to the component interfaces. In addition, if the component interfaces are coupled to other components, the components are typically not completely controllable through separate interfaces. This too can complicate the modeling and testing process. Consider the following conceptual representation of the set of components and interfaces shown in Figure 4.

To support a systematic verification approach that can be performed in stages where each component is completely verified with respect to the requirements allocated to it, the interfaces to the component should be explicitly and completely accessible, either using global memory, or better through get and set methods/procedures as reflected in Figure 4. For example, if the inputs to the B.2 component of higher-level component B are completely available for setting the inputs to B.2, and the outputs from the B.2 functions can be completely observed, then the functionality within B.2 can be completely specified and systematically verified. However, if interfaces from other components, such as B.1 are not accessible, then some of the functionality of the B.2 component is coupled with B.1, and the interfaces to B.2, must also include interfaces to B.1, or to other upstream components, such as component A. This interface coupling makes the test driver interfaces more complex to describe, but also forces the behavioral modeling to be described in terms of functionality allocated to combinations of components. The coupling reduces the reuse of components, and increases the regression testing effort due to the coupled aspects of the system components. The problems associated with testing highly coupled systems can be problematic for model-based testing, but also negatively impacts any type of testing. As discussed in Section 2.4, we have observed that interface-driven modeling has helped foster better system design by reducing the coupling, but also helps provide better support for testing.

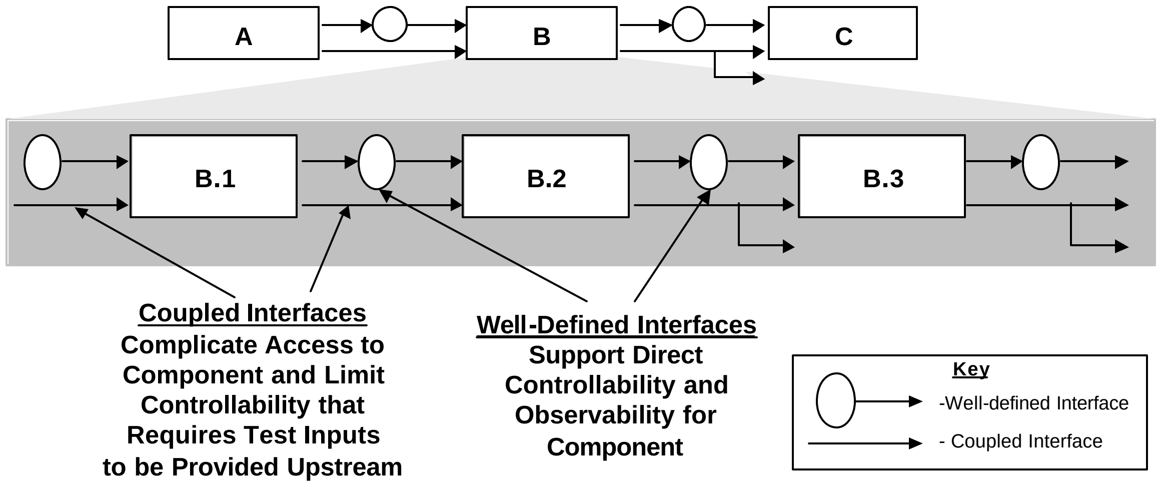

A
B
C
B.1
B.2
B.3
Coupled Interfaces Complicate Access to Component and Limit Controllability that Requires Test Inputs to be Provided Upstream
Well-Defined Interfaces Support Direct Controllability and Observability for Component
Key
-Well-defined Interface
- Coupled Interface

Figure 4. Conceptual Components of System

Systematic test coverage can typically be achieved directly from the verification model if the components of the system can be tested individually. Component integration testing can later be performed from higher-level models to ensure that the integration of the components (i.e., the contractual obligation of the integration) is systematically and completely verified.

2.4 Organizational Best Practices

Interface-driven modeling can be applied after development is complete, however, significant benefits have been realized when it is applied during development. Ideally, test engineers work in parallel with developers to stabilize interfaces, refine requirements, and build models to support iterative test and development. Test engineers write the requirements for the products (which in some cases are very poorly documented) in the form of models, as opposed to hundreds or thousands of lines of test scripts. They generate the tests vectors and test drivers automatically. During iterative development, if the component behavior, the interface, or the requirements change, the models are modified, and test cases and test drivers are regenerated, and re-executed. The key advantages are that testing proceeds in parallel with development. Users like Lockheed Martin state that test is being reduced by about fifty percent or more, while describing how early requirement analysis significantly reduces rework through elimination of requirement defects (i.e., contradiction, inconsistencies, feature interaction problems) [Saf00, KSSB01]. This typical and pragmatic use of TAF parallels eXtreme Programming (XP) [Bec99] where tests are created before the program. However, others refer to this model-based method as Extreme Modeling (XM) [PL01; BBWL00], which applies the principles to write tests prior to coding. With XP test code is developed manually, but with XM the requirements are modeled and tests are generated.

3 Example Requirements, Interfaces and Models

The Mars Polar Lander (MPL) project was launched by NASA on February 7, 1994. Six years later, on December 3, 1999, after the MPL had traveled over 35 million miles and was minutes away from its scheduled landing all contact with the craft was lost – and never regained. The MPL cost $165 million to develop and deploy. Without knowing anything about the actual cause of the problem, we modeled the Touchdown Monitor (TDM) requirements using the TAF tools, and were able to identify, in fewer than 24 hours, the software error associated with the MPL's landing procedures. Details relating to the failure scenario and the TDM component interfaces are shown in Figure 5.

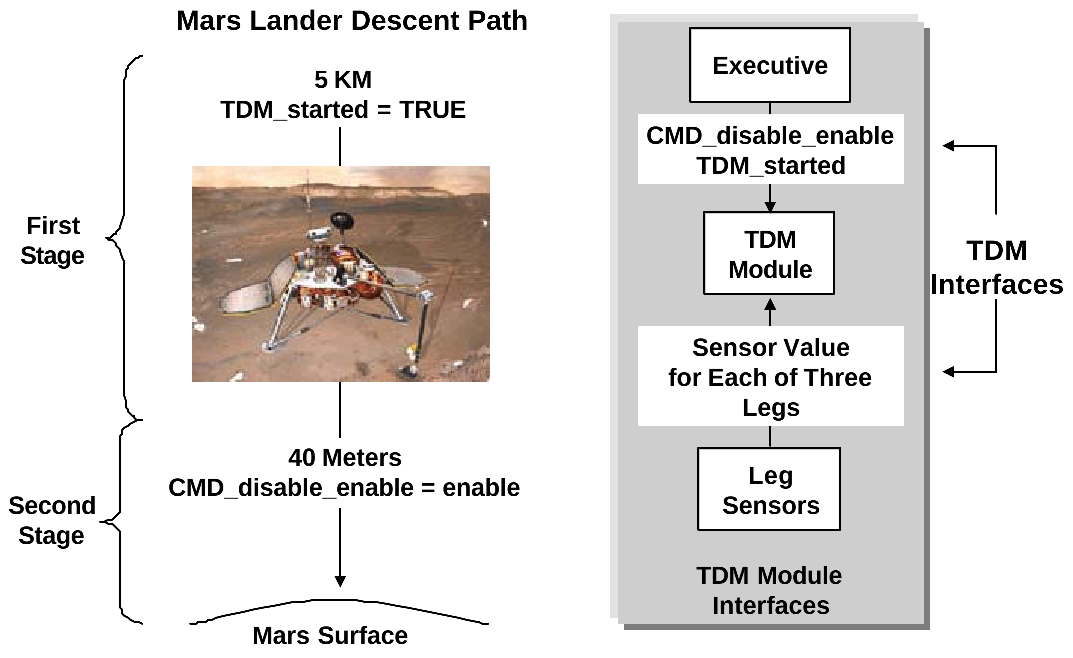


Figure 5. Mars Polar Lander Details

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 5, the TDM module is called by a real-time multi-tasking executive at a rate of 100 times per second, and receives information on the leg sensors from a second module. These two modules establish the interfaces to TDM. During the first stage, starting approximately five kilometers above the Mars surface, the TDM software monitors the three touchdown legs. There is one sensor for each leg that is used to determine if 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 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, starting about 40 meters above the surface, the TDM software was to monitor the remaining "good" sensors. When a sensor had two consecutive reads indicating touchdown, the TDM software was 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.

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 - leg 1, say - 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].` Since 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 Lander 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 upon 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.

After the fact, we had the opportunity to use TAF to see if it would have found the bug. We deliberately did not look at the code before creating our tests. Instead, we created them by modeling the English-language requirements in a tool based on the Software Cost Reduction method. Here are two of the actual requirements we used to develop our model:

> 3.7.2.2.4.2(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.
>
> 3.7.2.2.4.2(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.

Figure 6 shows a portion of the model. It controls the value of the model variable named `First_Marked_Bad`, which describes which leg was first discovered to be sending spurious touchdown indications. The value is shown in the final row of the table. The first row describes the variable's value before a clock tick event; the second describes the value afterwards.

Modes	**Condition**			
Before_event	TRUE	FALSE	FALSE	FALSE
Event_gen	NOT(TD_Sen1) AND NOT(TD_Sen2) AND NOT(TD_Sen3)	TD_Sen1	TD_Sen2	TD_Sen3
First_Marked_Bad=	0	1	2	3

Figure 6. Portion of the Mars Touchdown Monitor Model

The columns of the table describe the conditions under which `First_Marked_Bad` can have particular values. When the value of a cell is True, `First_Marked_Bad` has the corresponding value. Therefore `First_Marked_Bad` has the value 0 before the event, because the 0 column is the only one with a True value in the `Before_event` row. After the event, `First_Marked_Bad` has the value 1 if `TD_Sen1` is True (meaning the sensor on leg 1 has indicated touchdown in two consecutive clock ticks), 2 if `TD_Sen2` is True, 3 if `TD_Sen3` is True, and 0 otherwise.

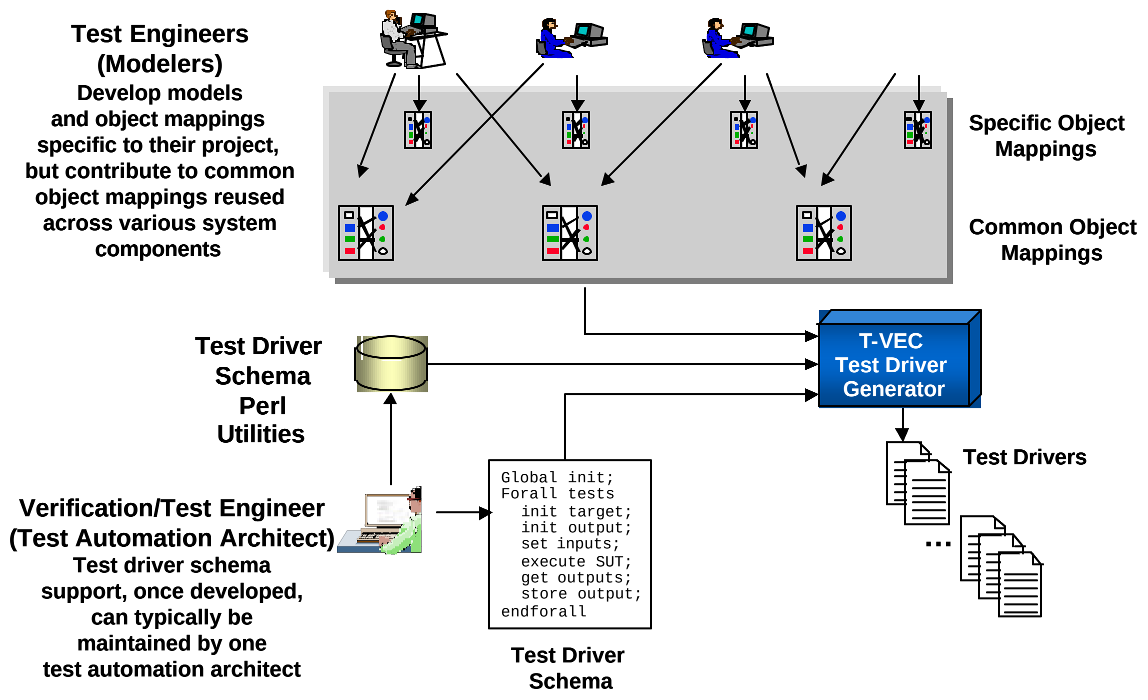
The full model consists of seven tables. The tables are linked together through intermediate variables like `First_Marked_Bad`. As is often the case for testing models, the model is simplified to reduce the number of test cases. In particular, the model assumes that only one leg will be marked bad at a time. This simplification still allowed the tests to find the problem. The tests were generated from the model and then test drivers were generated to execute the tests against the actual TDM code.

4 Organizational Impacts of Model-based Test Automation

The interfaces to the system under test are used in test driver development to map the model information to the actual system interfaces. This section provides a brief summary of the organizational impacts related to the use of lead test engineers in constructing test driver support. Often some test engineers do not have enough knowledge of the existing system component interfaces and lead test engineer knowledge is required to support the efforts. However, lead test engineers are usually a limited resource. This section explains how organizations have used this model-based test automation while leveraging scarce lead test engineer resources. In addition, there is a brief description of a recommended structure that supports reuse of the test driver information. Some details of test driver development are beyond the scope of this paper, because test drivers mappings are directly related to the test driver language and test environment, but detailed example models and associated test drivers in C++, Java, SQL, and Perl are available for download from: http://www.software.org/pub/taf/Reports.html.

4.1 Organizational Roles

In many test organizations, each test engineer is typically required to design test cases, write the test drivers for the test cases, execute the tests, and perform results analysis to determine if the test cases pass or fail. Understanding the details of the test environment typically requires some key skills and knowledge about the APIs of the system under tests. While applying the TAF method, as shown in Figure 7, the roles can be separated into test engineers that primarily develop models from the requirements (Modelers), and test engineers that develop the infrastructure for generating test drivers (Test Automation Architects). A test automation architect is usually a more knowledgeable developer with programming skills, an understanding of the system APIs, and knowledge of the test environment. The test automation architect typically develops libraries to automate test driver generation for the tests. The test automation architect also develops the test driver schema, which is tailored to the specific test environment. Lastly, the test automation architect typically develops and organizes object mappings, which map model variables to the APIs of the components under tests. These common object mappings provide the direct link between the interfaces of the system and the models. Modelers then use the common object mappings as a vocabulary (or glossary) of model variable to model the requirements. Time and effort are saved when the common object mappings are reused for many tests. Modelers may develop a few specific object mappings for their tests, as needed, however it is common for such specific mappings to be added to various common object mappings.


Test Engineers (Modelers)
Develop models and object mappings specific to their project, but contribute to common object mappings reused across various system components
Specific Object Mappings
Common Object Mappings
Test Driver Schema Perl Utilities
T-VEC Test Driver Generator
Test Drivers
...
Verification/Test Engineer (Test Automation Architect)
Test driver schema support, once developed, can typically be maintained by one test automation architect
Test Driver Schema

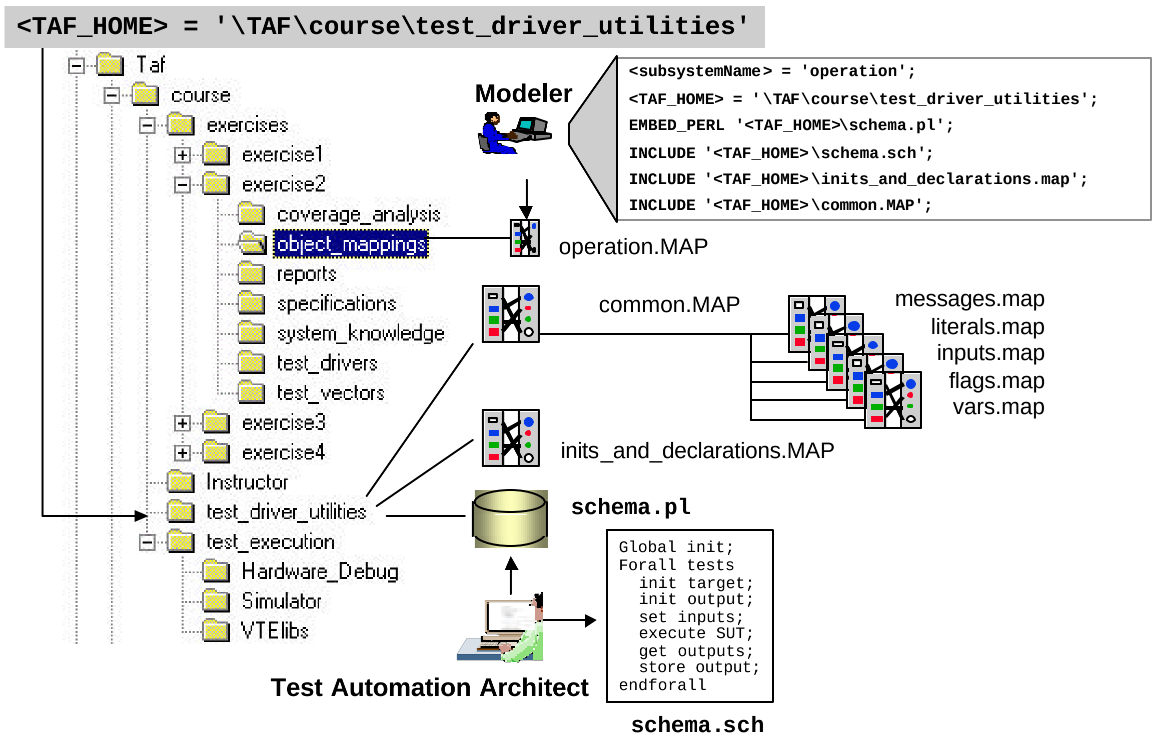
```
Global init;
Forall tests
   init target;
   init output;
   set inputs;
   execute SUT;
   get outputs;
   store output;
endforall
```

Figure 7. Separating Roles in Test Driver Development and Leverages Key Resources

We have worked with organizations and observed that one test automation architect can support the infrastructure development for several product environments with many modelers. This provides significant leverage of key resources. In addition, organizations that are developing object mappings in parallel to development have significantly improved the APIs to support test automation in the newer products by working with the design engineers early during the product development.

4.2 Test Driver Organization and Structure

Figure 8 provides a generic example of how the object mapping process works. This example is taken from a TAF training course. Test automation architects would set up a directory structure like the one shown at the left of Figure 8, where groups of related exercises use the test driver schema and common mappings that are located in the test_driver_utilities subdirectory. One exercise contains one or related models of related requirements. The modeler creates an object mapping file that references the relative directory path to the test_driver_utilities subdirectory. The common mapping file (common.MAP) also includes other types of mappings, like messages, literals, inputs, variables, etc. This allows the modeler to simply reference commonly developed object mappings. If there are changes in the system APIs, the common mappings, located in one place, can be updated, and related models and their associated tests can be regenerated for the newly added information. In addition, new test engineers will not fully understand the initialization and declaration information that is required within a test driver. Often these engineers start from existing test drivers and copy declaration and initialization into their test driver. With the TAF method, the test automation architect develops the initialization and declarations in a mapping file that can be included by the team of modelers. If this information changes, it is updated in one place and re-applied to other tests.



```
<subsystemName> = 'operation';
<TAF_HOME> = '\TAF\course\test_driver_utilities';
EMBED_PERL '<TAF_HOME>\schema.pl';
INCLUDE '<TAF_HOME>\schema.sch';
INCLUDE '<TAF_HOME>\inits_and_declarations.map';
INCLUDE '<TAF_HOME>\common.MAP';
```

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

Figure 8. Test Driver Organization Facilitates Reuse and Leverages Expertise

This structure allows test driver infrastructure to be reused across a project and can be managed by a test automation architect. It also provides a single linkable reference for modelers to reuse common information to support test driver generation. It facilitates regeneration when infrastructure changes are impacted by design changes to the actual target system. Test drivers can also be regenerated for regression testing to ensure that the system still operates properly, without requiring modelers to make changes, unless the specific requirements have changed.

5 Summary

This paper provides pragmatic guidance for combining interface analysis and requirement modeling to support model-based test automation. The model-based testing method and tools described in this paper have been demonstrated to significantly reduce cost and effort for performing testing, while also being demonstrated to identify requirement defects that reduce costly rework. These recommendations for defining interfaces that provide better support for testability are valid for all forms of testing. Organizations can see the benefits of using interface driven model-based testing to help stabilize the interfaces of the system early, while identifying common test driver support capabilities that can be constructed once and reused across related tests. In addition, parallel development of verification modeling is beneficial in development and helps identify requirement defects early to reduce rework. This concept has been characterized as eXtreme modeling, which is similar to eXtreme programming.

6 References

[Asi02] Aissi, S.,Test Vector Generation: Current Status and Future Trends, Software Quality Professional, Volume 4, Issue 2, March 2002.

[Bec99] Beck, K., Extreme Programming Explained: Embrace Change. Addison Wesley, 1999.

[BBN01a] Blackburn, M.R., R.D. Busser, A.M. Nauman, Removing Requirement Defects and Automating Test, STAREAST, May 2001.

[BBN01b] Blackburn, M. R., R.D. Busser, A.M. Nauman, How To Develop Models For Requirement Analysis And Test Automation, Software Technology Conference, May 2001.

[BBN01c] Blackburn, M. R., R.D. Busser, A.M. Nauman, Eliminating Requirement Defects and Automating Test, Test Computer Software Conference, June 2001.

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

[BBNKK01] Blackburn, M.R., R.D. Busser, A.M. Nauman, R. Knickerbocker, R. Kasuda, Mars Polar Lander Fault Identification Using Model-based Testing, Proceeding in IEEE/NASA 26th Software Engineering Workshop, November 2001.

[BBN01d] Busser, R. D., M. R. Blackburn, A. M. Nauman, Automated Model Analysis and Test Generation for Flight Guidance Mode Logic, Digital Avionics System Conference, 2001.

[BBWL00] Boger, M., T. Baier, F. Wienberg, and W. Lamersdorf. Extreme modeling. In Proc. Extreme Programming and Flexible Processes in SW Engineering (XP'00), 2000.

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

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

[KSSB01] Kelly, V. E.L.Safford, M. Siok, M. Blackburn, Requirements Testability and Test Automation, Lockheed Martin Joint Symposium, June 2001.

[PL01] Pretschner, A., H. Lotzbeyer, Model Based Testing with Constraint Logic Programming: First Results and Challenges, Proc. 2nd ICSE Intl. Workshop on Automated Program Analysis, Testing and Verification (WAPATV'01), Toronto, May 2001.

[PM91] Parnas, D., J. Madley, Functional Decomposition for Computer Systems Engineering (Version 2), TR CRL 237, Telecommunication Research Inst. of Ontario, McMaster University, 1991.

[Rob00] Robinson, H., http://www.model-based-testing.org/.

[Rom85] Roman, G.C., A Taxonomy of Current Issues in Requirements Engineering, IEEE Computer, 18(4):14-23, 1985.

[RR00] Rosario, S., H. Robinson, Applying Models in Your Testing Process, Information and Software Technology, Volume 42, Issue 12, 1 September 2000.

[Sch90] van Schouwen, A.J., The A-7 Requirements Model: Re-Examination for Real-Time System and an Application for Monitoring Systems. TR 90-276, Queen's University, Kinston, Ontario, 1990.

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

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

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

[YZCG84] Yeh, R.T., P. Zave, A.P. Conn, G.E. Cole, Software Requirements: New Directions and Perspectives, in Handbook of Software Engineering, Editors C. R. Vick and C. V. Ramamoorthy), Van Nostrand Reinhold, 1984.