Why Model-Based Test Automation is Different and What You Should Know to Get Started

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

Test engineers, their managers, as well as the project developers often have many different views and misconceptions about tools and methods that provide test automation support. This paper compares model-based testing with three other generations of test automation. Model-based test automation can be considered a fourth generation test automation. It supports defect prevention, early requirement defect identification, and automatic generation of tests from models, which eliminates manual test design and reduces cost. However, to be effective at adopting model-based test automation, there are specific skills that will be required to incorporate this type of test automation into an organization. The paper discusses the organizational, personnel, and development lessons learned from working with numerous companies and projects over the past several years. It recommends how to get started, how to select a project, and how to organize a project. Finally, the paper describes how to measure, track and estimate the project completion date during the first non-pilot project.

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

1. Introduction

The increased complexity of systems as well as short product release schedules makes the task of testing challenging. One of the key problems is that testing typically comes late in the project release cycle, and traditional testing is performed manually. When bugs are detected, the cost of rework and additional regression testing is costly and further impacts the product release. The increased complexity of today's software-intensive systems means that there are a potentially indefinite number of combinations of inputs and events that result in distinct system outputs, and many of these combinations are often not covered by manual testing. We work with companies that have high process maturity levels, and excellent measurement data that shows that testing is more 50-75% of the total cost of a product release, yet these mature processes are not addressing this costly issue.

As Fewster and Graham point out, test tools may not replace human intelligence in testing, but without them testing complex systems at a reasonable cost will never be possible [FG99]. There are commercial products to support automated testing, most based on capture/playback mechanisms, and organizations that have tried these tools quickly realize that these approaches are still manually intensive and difficult to maintain. Even small changes to the application functionality or GUI can render a captured test session useless. But more importantly, these tools don't help test organizations figure out what tests to write, nor do they give any information about test coverage of the functionality.

Section 2 of this paper provides a perspective on various approaches to test automation. Section 3 describes how model-based testing is different from other forms of test automation. Section 4 discusses why testing needs to be treated as an engineering activity. It should start early with requirement analysis to better understand the testability of the requirements, and the design should

Copyright © 2004, Software Productivity Consortium, NFP. All rights reserved.

support testability, which directly impacts viability of test automation. Through test automation, testing can be more systematic, provide greater coverage, and reduce manual cost and effort. Section 5 discusses how pilot projects help an organization get started with model-based testing. Section 6 briefly describes how model-based testing provides objective measures derived from requirement and interface information to support early project estimation and continuous tracking.

2. Approaches to Test Automation

Unfortunately, a significant amount of testing is still performed manually. This test creation process is error-prone, with testers sometimes unintentionally repeating cases while leaving others untested. This requires manual effort to perform the following test-related activities:

- Test design – determination of the test cases to cover the requirements of the SUT
- Test execution - manual entry of test cases and associated data, primarily through a client (sometimes GUI or web-browser) interface
- Test coverage – manual analysis to ensure that all combinations of logic are tested, which requires significant human expertise (domain expertise) and time
- Test results analysis – manual analysis to check that the actual output (outcomes) of the SUT are equivalent to the expected outputs (outcomes)

This section discusses various approaches to test automation, primarily focused on test design and test execution, as these are the most time consuming testing activities. The test automation often depends on the testability of the system under test (SUT). Design for testability is discussed in Section 4.

2.1. Test Scripting

All automated test execution is based on some form of test script (aka test driver) that can run automatically without human interaction. Test scripts are programs that provide general mechanisms that support other test automation approaches, including capture/playback, hybrids that extend capture/playback, as well as test generation approaches. Test scripts can be developed using standard application languages, such as VB, C, C++, Java, Perl, specialized languages such as Tcl, and Python, or custom languages supported by testing tools. Test scripting usually requires hooks provided through testable designs, that can include an application programming interface (API), a component interface (e.g. COM), a protocol interface (e.g., HTTP), or a debugging or emulation interface. Some testing tools provide libraries or support that leverage API, component, and protocol interfaces.

For functional testing, test scripts typically have a pattern, as reflected in Figure 1 that:

- Initializes the SUT
- Loops through a set of test cases, and for each test case
    - Initialize the target [optional]
    - Initializes the output to a value other than the expected output (if possible)
    - Sets the inputs
    - Executes the SUT

Copyright © 2004, Software Productivity Consortium, NFP. All rights reserved.

- Captures the output and stores off the results to be verified against the actual output at some later time, when a test report can be created

Most of the following approaches are built on some type of test scripting.

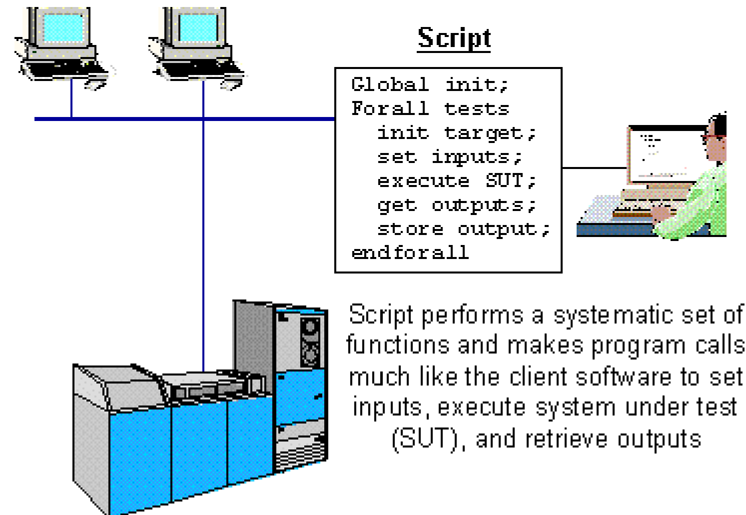

Figure 1. Test Script Automation

2.2. Capture/Playback Approach

Capture/playback tools capture sequences of manual operations in a test script that are entered by the test engineer. Although the underlying mechanisms of capture/replay rely on test scripts, most users of the tools operate these tools using the recording feature of the tool strictly from a GUI point-of-view, and do not work at the scripting level. The capture/playback approach still relies on people to manually determine and enter the test cases, similar to the manual testing approach. The benefit of this approach is that the captured session can be re-run at some later point in time to ensure that the system performs the required behavior.

However, there are shortcomings to capture/playback.

- When the system functionality changes, the capture playback session will need to be completely re-run to capture the new sequence of user interactions. The re-capturing process is still manual and typically takes as long to perform each additional time as it did the first time.
- Capture/playback tools are supposed to recognize GUI objects, even if the GUI layout has changed. However, this is not always the case, because the effective use of capture/playback tools often depends on visibility of GUI object, and naming conventions, and this requires support during the GUI design and implementation phases.
- The appropriate selection of the object-recording mode versus the analog recording mode, and synchronization of the replay is vital to playback for regression testing.
- Web sites are increasingly complex, and manually recording a sample set of testing scenarios with a capture/playback tool can be very time-consuming. Due to the limited schedules, it is nearly impossible to record more than a few possible paths, and Web-

Copyright © 2004, Software Productivity Consortium, NFP. All rights reserved.

site test coverage using capture/playback tools ends up being typically limited to a small portion of the Web-site functionality.

- More advanced capture/playback tools often provide some level of abstraction when recording user actions and increased portability of test scenarios (for instance, by recording general browser actions instead of mouse actions on specific screen coordinates), but changes in the structure of a Web site may prevent previously recorded test scenarios from being replayed, and hence may require re-generating and re-recording a new set of test scenarios from scratch [BFG02].
- Many of the capture/playback tools provide a scripting language, and it is possible for engineers to edit and maintain such scripts, but this does require the test engineer to have programming skills.

2.2.1 Test Abstraction and Data Driven Approach

There are related approaches that extend capture/playback such as data-driven, action-based, keyword-based, object-based, and class-based. Each of these approaches still require testers to interpret the requirements and design the test cases to cover each requirement, and requires some engineer with development skill to define the test scripts that are associated with the test abstraction mechanisms for accessing information from a data-driven repository, such as a database, in addition to implementing actions and associated parameters, object, or class-based operations. The approaches categorized in Table 1 use an abstraction mechanism (e.g. action word) defined by the test writer to define test cases at a higher-level of abstraction than the underlying test script.

Table 1. Categories of Test Abstraction

Category	Abstraction Mechanism	Test Development
Action Word-based	Actions mapped to scripts	Application experts combine actions for testing often with data sets
Window-based	Display pages/windows mapped to input set and output	Test scenarios combine windows and data sets
Object-based	Script functions mapped to application objects	Test sequences on objects are developed by combing script functions
Class-based	Scripts mapped to actions performed against a class of objects	Test sequences on objects are developed by combing script functions

For action words, there is typically an action word and a set of parameters (i.e., data values) that are associated with the action word. To implement the test, there is a mapping for each action word to an associated script that takes the various parameters that are applicable to that action word to carry out the test.

Copyright © 2004, Software Productivity Consortium, NFP. All rights reserved.

The window-based approach derives from the capture/playback tools where the widgets (e.g., fields, menus, etc.) are associated with parameter values that can be represented in a spreadsheet. Testers define scenarios of windows, and associated sets of parameter values to cover various test cases.

Object-based provides the object-oriented equivalent of the action-word based, where the test scripts associated with the data values for testing the object are mapped to the attributes and methods that must be carried out to involve the objects in test case scenarios.

Class-based approaches define one class, with the associated test scripts that can be used for one or more objects of that class.

2.3. Model-Based Test Automation

There are various approaches to model-based test generation that based on various modeling forms, such as state machines, functional tabular condition/action models, control system models, language models, and hybrids. Once the tests are generated, they can be transformed into test scripts that can execute the tests. They key advantage of this technique is that the test generation can systematically derive all combination of tests associated with the requirements represented in the model to automate both the test design and test execution process.

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. There are various approaches to model-based testing and Robinson hosts a website that provides useful links to authors, tools and papers [Rob00].

3. What Makes Model-Based Test Automation Different?

We have applied 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 (SCR) method, as well as methods that focus on representing design information, like MathWorks' Simulink or National Instruments' MATRIXx, which supports control system modeling for automotive and aircraft systems.

The TAF approach, as illustrated in Figure 2 reflects how modelers develop the logic associated with requirements for data and control processing of the SUT using models. The process involves three roles including requirement engineer, designer/implementer, and tester (modeler). A requirements engineer typically documents the requirements in text-based documents. A designer develops the system/software architecture, design, components, and interfaces. Although it is common to start the process with poorly defined requirements, inputs to the process can include system and software requirement specifications, user documentation, interface control documents, application program interface (API) documents, previous designs, and old test scripts. Testers (modelers) use any available information to clarify the requirements in the form of a model, which specifies behavioral requirements in terms of the interfaces for the SUT (or component under test).

Copyright © 2004, Software Productivity Consortium, NFP. All rights reserved.

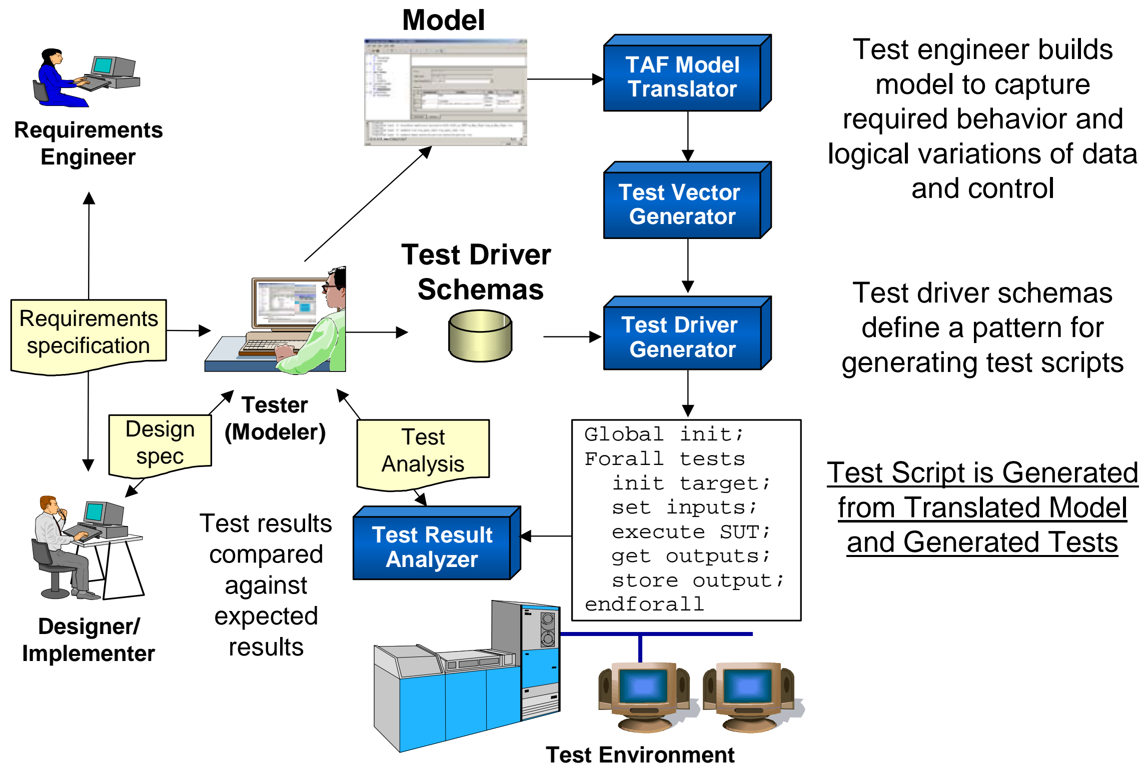

Figure 2. Model-Based Test Automation

Models are translated, and T-VEC, the test generation component of TAF produces 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 (test scripts). The test drivers are then executed against the implemented system during test execution. The test execution analysis compares automatically the actual outputs against the expected outputs and produces a test results report.

3.1. Benefits of Model-based Automated Testing

Model-based approaches, like TAF, leverages models to support requirement defect analysis and to automate test design. Model checking can ensure properties, like consistency, are not violated. In addition model helps refine unclear and poorly defined requirements. Once the models are refined, tests are generated to verify the SUT. Eliminating model defects before coding begins, automating the design of tests, and generating the test drivers or scripts results in a more efficient process, significant cost savings, and higher quality code. Some other advantages of this approach include:

- All models can use the same test driver schema to produce test scripts for the requirements captured in each model. Many test driver schemas already exists for languages such as C, C++, VB, Java, Perl, SQL, PLI, JCL, Ada, XML, HTML, JDBC, ODBC, WinRunner, DynaComm, and various Simulators

Copyright © 2004, Software Productivity Consortium, NFP. All rights reserved.

- When system functionality changes or evolves, the logic in the models change, and all related test are regenerated using the existing test driver schema
- If the test environment changes, only the test driver schema needs modification. The test drivers associated for each model can be re-generated without any changes to the model

These results are common among users of model-based testing approaches [RR00; KSSB01; BBNKK01; BBN01d; Sta00; Sta01]. The initial expectation is that model-based testing supports automated test generation, but the unexpected benefit achieved is better understanding of the requirements, improved consistency, completeness, and most importantly, early requirement defect identification and removal. These benefits are briefly discussed below.

3.1.1 Comprehensive Tests

TAF uses the model to traverse the logical paths through the program, determining the locations of boundaries and identifying reachability problems, where a particular thread through a model may not be achievable in the program itself. TAF uses test selection criteria based on domain testing theory [WC80] to select the test inputs that are most likely to identify faults in the program. Domain testing theory is based on the intuitive idea that faults in implementation are more likely to be found by test points chosen near appropriately defined program input and output domain boundaries [TVK90].

3.1.2 Improved Requirements

In order to be testable, a requirement must be complete, consistent and unambiguous. While any potential misinterpretation of the requirement due to incompleteness is a defect, TAF focuses on another form of requirement defect, referred to as a contradiction or feature interaction defect. These arise from inconsistencies or contradictions within requirements or between them. Such defects can be introduced when more than one individual develops or maintains the requirements. Often the information necessary to diagnose requirement contradictions spans many pages of one or more documents. Such defects are difficult to identify manually when requirements are documented in informal or semi-formal manners, such as textual documents. Although rigorous manual inspection techniques have been developed to minimize incompleteness and contradictions, there are practical limits to their effectiveness. These limits relate to human cognition and depend on the number and experience of people involved. TAF supports requirement testability analysis, which allows developers to iteratively refine and clarify models until they are free of defects.

3.1.3 Defect Discovery

Defect discovery using model-based test automation is both more effective and more efficient than using only manual methods. One pilot study, conducted by a Consortium member company, comparing formal Fagan inspections with TAF requirement verification, revealed that Fagan inspections uncovered 33 requirements defects. In comparison, TAF uncovered all 33 of the Fagan inspection defects plus 56 more. Attempting to repeat the Fagan inspection did not improve the Fagan inspection's return on investment. The improved defect detection of TAF prevented nearly two-thirds more defects from entering the rest of the lifecycle.

Copyright © 2004, Software Productivity Consortium, NFP. All rights reserved.

Similar results where measured by Rockwell Collins who used a requirement modeling method to develop the mode control logic of a Flight Guidance System (FGS) avionics system, and later used an early version of TAF for model-based analysis and test automation. As reflected in Figure 3, the FGS was first specified by hand using the Consortium Requirement Engineering Method (CoRE). It was then inspected, and about a year later entered into a tool supporting the SCR method provided by the Naval Research Laboratory (NRL). Despite careful review and correction of 33 errors in the CoRE model, the SCRtool's analysis capabilities revealed an additional 27 errors. Statezni, a Rockwell engineer, later used an early TAF translator and the T-VEC toolset to analyze the SCR model, generate test vectors and test drivers. The test drivers were executed against a java implementation of the FGS requirements and revealed six errors. Offutt applied his tool to the FGS model and found two errors, and the latest TAF toolset, identified 25 model errors [BBN01d].

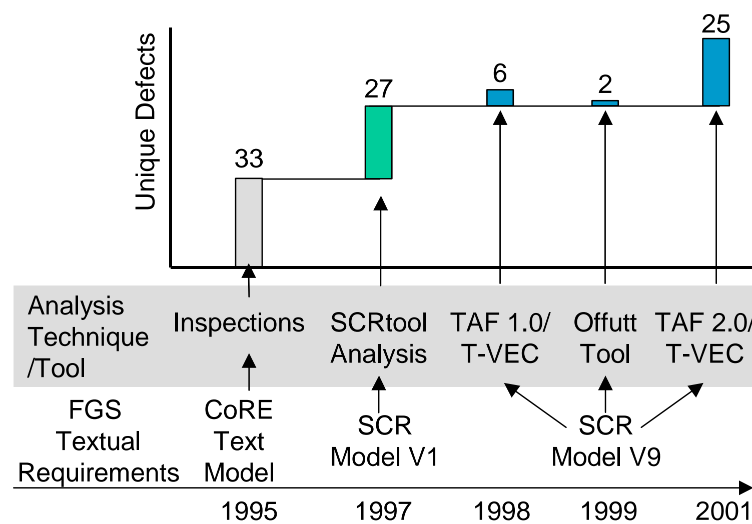

Figure 3. Comparison of Defect Discovery by Tools and Methods

Following manual test generation practices, defects are not identified until late in the process, sometimes after release, when they are most expensive to fix. Automating test generation based on models, defects are found earlier in the process and faster. The rate of defect discovery increases early in the process, but quickly curtails. Many defects are found in the requirements phase, before they propagate to later development phases. Defect prevention is most effective during the requirements phase when it costs two orders of magnitude less than after the coding process.

Figure 4 represents the conceptual differences between manual and automatic test generation. The existing process of discovering and eliminating software defects is represented by the curve labeled "Old" while the effects of early defect discovery aided by automation is illustrated by the trend curve labeled "New." Industrial applications have demonstrated that TAF directly supports early defect identification and defect prevention through the use of requirement testability analysis. The structured process of modeling supports defect prevention by eliminating processes that have in the past allowed defect-types to be repeatedly introduced into products.

Copyright © 2004, Software Productivity Consortium, NFP. All rights reserved.

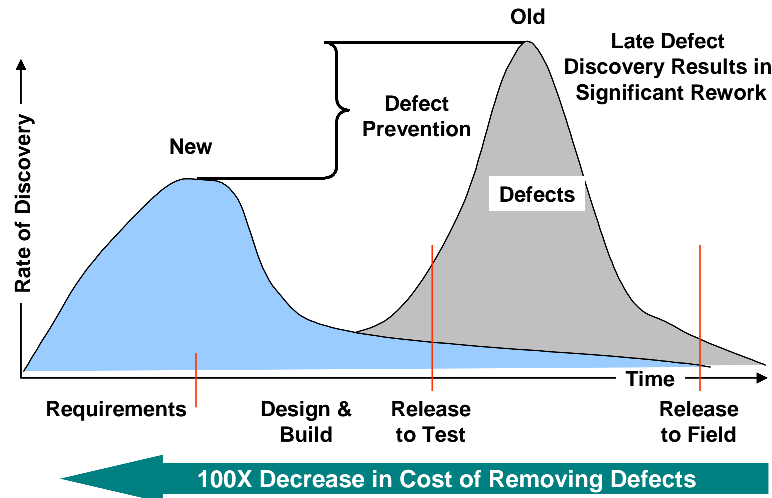

Figure 4. Member Savings

3.2. Requirement Validation

Requirement validation ensures captured requirements reflect the functionality desired by the customer and other stakeholders. Although requirement validation does not focus specifically on requirement testability analysis, it does support it. Requirement validation involves an engineer, user or customer judging the validity of each requirement. Models provide a means for stakeholders to precisely understand the requirements and assist in recognizing omissions. Tests automatically derived from the model support requirement validation through manual inspection or execution within simulation or host environments.

4. Process and Organizational Impacts of Model-Based Testing

Model-based testing is not just a better way to test, but can spawn organizational impacts that promote a more continuous test process, as well as system architecture impacts that improve the overall system design. The most effective approach that we have used to foster better design for testability is to use a continuous testing process, where test engineers are involved early in the requirement analysis to ensure that the design has interfaces for testing.

4.1. Design for Testability

There are many benefits derived from performing design for testability, including more improved test coverage, simpler tests design, enhanced fault analysis (debugging), as well as more options for test automation. Without it, most tests must be performed manually. The concepts of design for testability have been around for many years. By creating infrastructure support within the application, and specifically to the interfaces of the SUT, we can support three notions [WP82]:

1. Predictability of the tests, which supports a means for conclusively assessing whether the GUI performed correctly or not.
2. Controllability of the tests, which permits the test mechanism to provide inputs to the system and drive the execution through various scenarios and states to foster the need for systematic coverage of the GUI functionality

Copyright © 2004, Software Productivity Consortium, NFP. All rights reserved.

3. Observability of the system outputs that can lead to a decision as to whether the outputs are desirable (correct) or faulty.

Design for testability provides the sum of these features to test execution, but with the support of these features permits more automated types of test automation to be used for test execution. As shown in Figure 5, it is important for the design engineers to expose some of the internals of the SUT, like component interfaces, to provide more controllability and observability of internal information that passes into and out of the system through program-based interfaces. It is best to have early interaction between the lead testers and the lead designers so that the program-based interfaces that support testability are exposed.

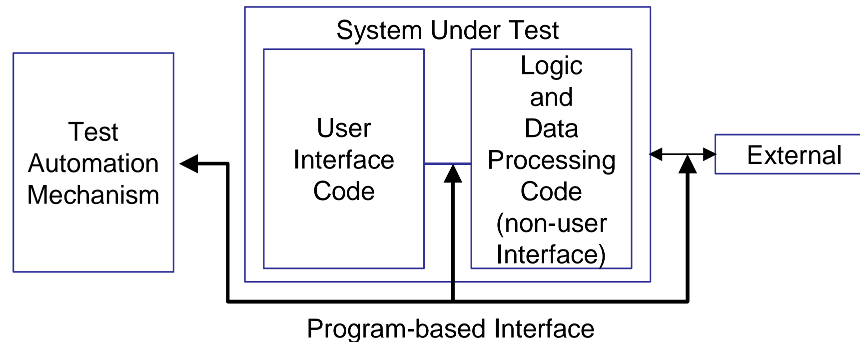

Figure 5. Program-Based Interfaces to SUT

Design for testability should occur at many or all of the layers of the software system architecture because it results in less coupling of the system components. If the component interfaces are coupled to other components, the components are typically not completely controllable through separate interfaces. This can complicate the modeling and testing process. Consider the following conceptual representation of the set of components and interfaces shown in Figure 6.

To support systematic testing that can be performed in stages where each component is completely tested 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 6. 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 tested. 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 forces the 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.

Copyright © 2004, Software Productivity Consortium, NFP. All rights reserved.

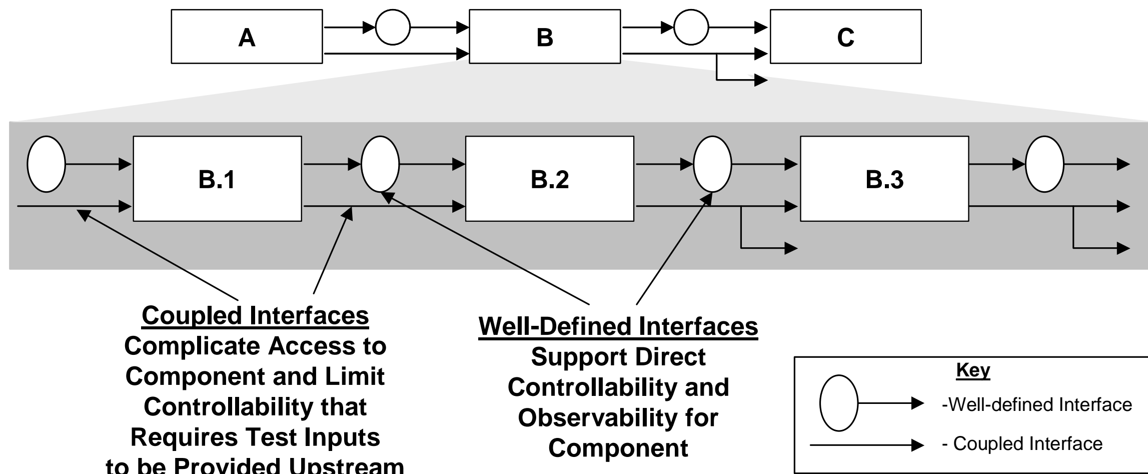

Figure 6. Conceptual Components of System

4.2. Interface-Driven Requirement Modeling

The recommend process identified by using TAF with companies over the last several years is reflected in Figure 7, which provides a more detailed perspective of the modeling process flow as it relates to time. This perspective extends Figure 2 with some additional details. The requirement engineer and design engineers work in parallel with the test engineer to define requirements in terms of component interfaces. This drives the design to identify the component interfaces early in the process to help stabilize the architecture, which providing the interface information that is required to support test driver generation. The term verification model is used to refer to the requirements of a component that are defined in terms of the interfaces. The test engineer modeler typically developed incrementally and uses TAF tools to perform model analysis, and correct any inconsistency in the requirements very early in the process.

Once the models are correct, test drivers can be generated. A second type of test engineer, called the Automation Architecture, typically develops the test driver schema for the particular testing environment, usually from one of the existing schemas. Usually, one test automation architect can support 20 or more modelers. Once a test driver schema is created, it doesn't change very much. The interfaces from the verification model are mapped to the APIs of the implementation using object mappings, and then test drivers are generated to support automated test execution. Often test drivers are available for the implementer to use before the implementation is completed. This has the side benefit of reducing the unit testing that the developer typically performs. If test failures are identified, each test has requirement-to-test traceability information that allows failures to be traced backwards to the requirement. This allows fault analysis information to determine if the requirements or the implementation are incorrect.

Copyright © 2004, Software Productivity Consortium, NFP. All rights reserved.

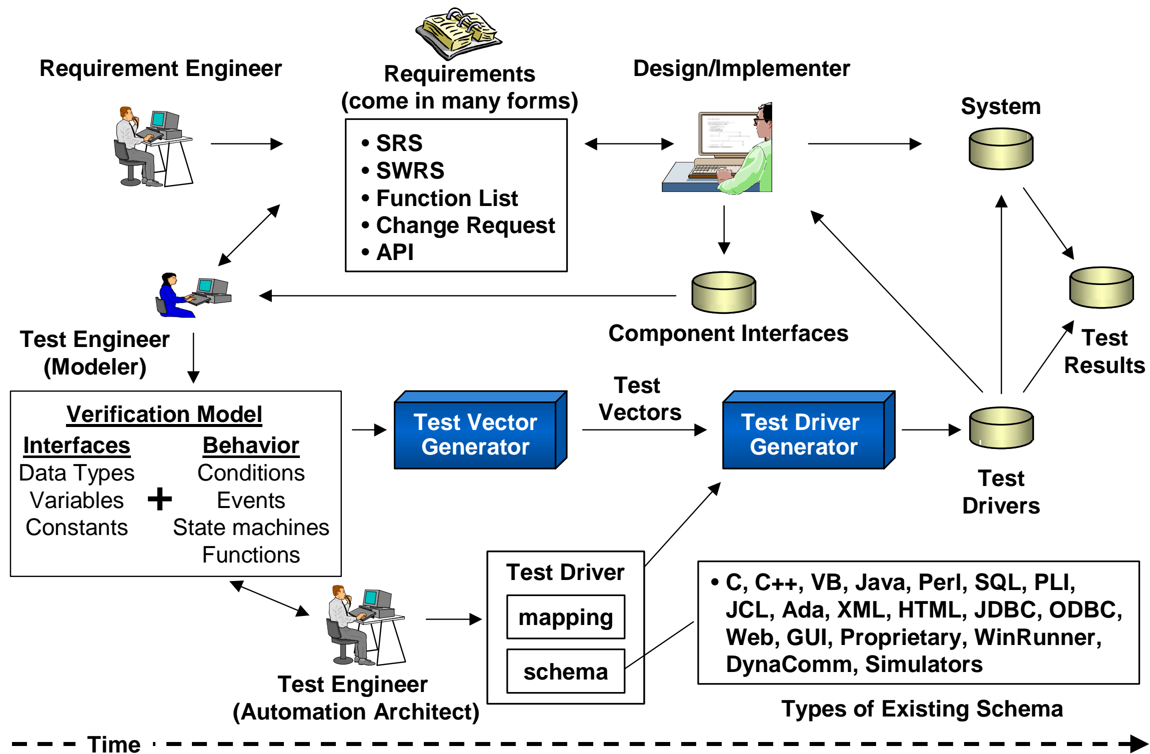

Figure 7. Process Summary of Roles versus Time

5. Get Started With A Pilot Project

Before selecting a test automation approach for your project it is useful to use a small pilot project to better understand the capabilities of the test automation tools, to understand how they work with the development tools and architecture, and to access how the tools fit into the existing process. We have found it useful to use two types of pilot projects in deploying model-based testing.

5.1. Three-day Mini Pilot

A three-day mini-pilot project uses a modeling expert to work onsite with a small team that includes someone that can represent the roles of the requirement engineer, the design engineer, and test engineer. The objectives of this mini-pilot are to quickly model some part of a product application, tailor an existing test driver schema to generate the test drivers, and execute the test drivers through some existing system. This is important to show all members of the team, including some management sponsor, that it is feasible to use in their product areas. During the remainder of the three-day pilot, the model can be evolved to show how the test driver schema can be reused. Mini-pilot projects are useful to:

- Demonstrate feasibility and time-reduction implications of applying models and reusable test driver schemas
- Work with developers through hands-on use
- Use experiments to investigate applicability on different types of projects and applications
- Review existing approach(es) and make recommendations for improvement and use of models

Copyright © 2004, Software Productivity Consortium, NFP. All rights reserved.

- Determine feasibility for application in other potential areas where verification is needed to significantly reduce cycle time
- Determine the required skills of the participants for the follow-on pilot

The final steps of the mini-pilot are to decide on a small part of an actual feature that can be used in a pilot project that is conducted by the members of the project team, with support of the model-based expert. The last step is to provide an out-briefing to management that describes the follow-on plan for a small, less than three month, pilot project.

5.2. Pilot Project Candidate

The candidate project should develop a small thread of functionality for a new feature added to some component of the system. The feature should not be on the critical path in terms of schedule release of the product. The interfaces to the components with the new feature should be well defined, supporting testability as described in Section 4.1. In addition, the selected feature should be one that is likely to have continual changes or related extensions in the future. This will illustrate how to leverage existing models to support new features, as well as support full regression testing with a fraction of the typical effort. It is usually not necessary to have more than one or two engineers involved in the first pilot, but requirement and design knowledgeable engineers should be available to provide requirement and interface information to the pilot team. The key results of the pilot project should a better understanding of how to tailor the existing software development process for adoption by a larger team. It is recommended that the candidates selected for the project be used as the mentors for each follow-on projects.

6. Objective Measures

During the pilot project, and certainly after the first non-pilot project, there are some easy measures that can be used to track and estimate project completion dates. Figure 8 provides a perspective on the key measurement information associated with TAF model-based testing. These measures and their use are described in terms of an information model adapted from the ISO/IEC 15939, *Software Engineering - Software Measurement Process*. The key elements of the information model are ***attributes, base measures, derived measures, indicators,*** and ***information products.*** An attribute is a property or characteristic of a process or product that is the subject of measurement (e.g., requirement). A base measure or data primitive is a quantification of a single attribute. A derived measure combines two or more values of base measures using a mathematical function. An indicator can be 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. An information product consists of one or more indicators with corresponding interpretations. The information product forms the basis for action by the decision maker. These measures can be combined in many different ways to form indicators that satisfy specific information needs.

The TAF modeling approach results in four key base measures. The four key model-based measurement attributes associated with base measures are the requirements, modeled requirement threads, model variables, and object mappings. Requirement 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, and this results in the base measure number of variables. After model translation and processing, the model

Copyright © 2004, Software Productivity Consortium, NFP. All rights reserved.

requirements are converted into requirement threads, which is a base measure related to requirements. Finally, 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.

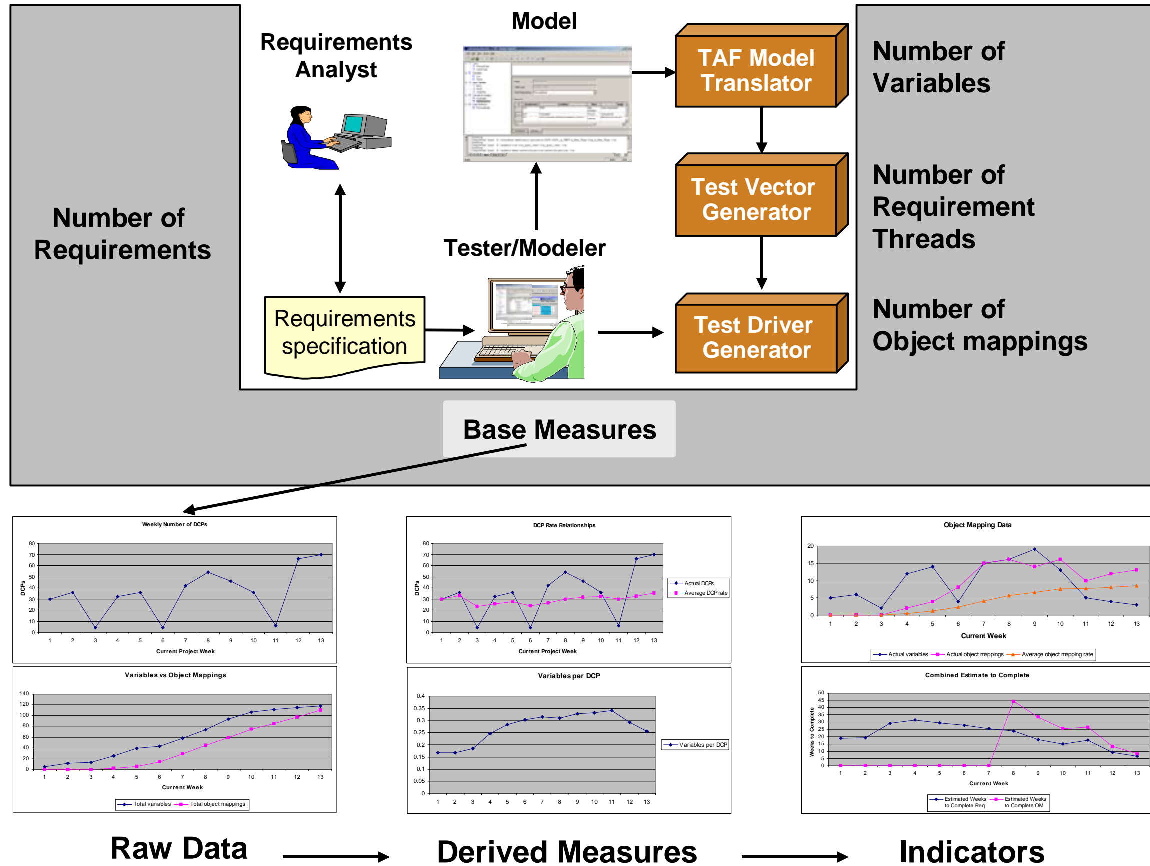

Figure 8. Process View of TAF Measurement

The base measures are combined into derived measures to produce indicators to provide information such as requirement modeling rate, requirement-to-test density, project duration, and estimated project completion. There are graphical representations of the base and derived measures, as well as formulas that use historical measures to predict project duration and usage of real-time project data to predict the completion of an ongoing project. This measurement-related information should help managers and project leads in predicting schedule duration and estimating project completion dates. Historical measurement information can be used prior to the start of a new project, but it also is important to use data derived during the project.

7. Summary

The first two generations of test automation focused on test execution, and had limited support for test design. The third generation test automation tools separate test scenario definition from test scripting-based automation, but still have limited support for test design. Model-based test automation, like the Test Automation Framework (TAF), can be considered a fourth generation test automation. It supports defect prevention, requirement defect identification, and automatic generation of tests from models, which eliminates manual test design and reduces cost. Model-

Copyright © 2004, Software Productivity Consortium, NFP. All rights reserved.

based test automation supports both requirement-based and design-based models. Early identification of requirement defects reduces rework involved in developing and testing both software and systems.

Model-based development also affects the organization. Development teams have reported significant cost and effort savings using approaches like TAF. Teams have found that requirement modeling takes no longer than traditional test planning, while reducing redundancy and building a reusable model library capturing the organization's key intellectual assets. Organizations can see the benefits of using interface driven model-based testing that includes design for testability to help stabilize the interfaces of the system early, while identifying component interfaces that support automated test driver generation that can be constructed once and reused across related tests.

Parallel development of modeling is beneficial in development and helps identify requirement defects early to reduce rework. Because testing activities occur in parallel to development efforts, testing teams get involved from the beginning and stay involved throughout the process, reducing the risk of schedule overruns. Defect prevention is a key benefit of the approach. It is achieved using model analysis to detect and correct requirements defects early in the development process. The verification models enable automated test generation. This eliminates the typically manual and error-prone test design activities and provides measurable requirement-based test coverage. Organizations have demonstrated that the approach can be integrated into existing processes to achieve significant cost and schedule savings.

The best way to get started is to use a pilot project to assess how to use model-based testing, and to best understand the organizational resources required to tailor the existing development process for a successful deployment on an actual project. TAF has been applied to applications in various domains including critical applications for aerospace, medical devices, IT applications including databases, client-server, web-based, automotive, telecommunication, and smart cards. Pilot demonstration can leverage existing test driver generation schemas supporting most any language (e.g., C, C++, VB, Java, Ada, Perl, PL/I, SQL, XML, etc.) as well as proprietary languages, COTS test injection products (e.g., DynaComm®, WinRunner®) and test environments.

Objective measurement support provides managers with tools to track and estimate project completion dates from the beginning of the first project with a few simple measures. Most users of the approach have reduced their verification/test effort by 50 percent.

8. References

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

[BFG02] Benedikt, M., J. Freire, P. Godefroid, VeriWeb: Automatically Testing Dynamic Web Site, http://www2002.org/CDROM/alternate/654/, Bell Laboratories, Lucent Technologies.

[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.

Copyright © 2004, Software Productivity Consortium, NFP. All rights reserved.

[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.

[FG99] Fewster, M., D. Graham, *Software Test Automation: Effective Use of Test Execution Tools. Addison-Wesley: Boston, MA., 1999.*

[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/.

[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.

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

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

[WP82] Williams, T. W., K. P. Parker, Design for Testability - A Survey, IEEE Trans. Comp. 31, pp. 2-15, 1982.

Copyright © 2004, Software Productivity Consortium, NFP. All rights reserved.