This paper was distributed at the Software Quality Week Conference in May, 1997.

Model Based Testing

Larry Apfelbaum
General Manager
603-879-3555
larry@sst.teradyne.com

John Doyle
Support Manager
603-879-3499
jd@sst.teradyne.com

Teradyne Software & Systems Test
44 Simon Street
Nashua, NH 03060
www.teradyne.com/sst

Abstract

The use of a model to describe the behavior of a system is a proven and major advantage to test development teams. Models can be utilized in many ways throughout the product life-cycle, including: improved quality of specifications, code generation, reliability analysis, and test generation. This paper will focus on the testing benefits and review some of the historical challenges that prevented model based testing and present the solutions that overcame these challenges. In addition benefits of a model based approach are reviewed in the context of two real applications, a call processing feature and a UI of a workflow oriented database system.

Outline:

1. Rationale for modeling

Models are used to understand, specify and develop systems in many disciplines. From DNA and gene research to the development of the latest fighter aircraft, models are used to promote understanding and provide a reusable framework for product development. In the software engineering process, models are now accepted as part of a modern object oriented analysis and design approach by all of the major OO methodologies. Papers and books have been written about the application of models to test development and reliability analysis for over two decades; however, except for leading edge companies, test creation is still a manual process with few quantitative metrics and low reuse. The objective of this

paper is to present an approach for test creation based on a graphical model that describes the behavior of the system to be tested; and a set of heuristics that enable tests to be generated from that model.

Modeling is a very economical means of capturing knowledge about a system and then reusing this knowledge as the system grows. For a testing team, this information is gold; what percentage of a test engineer's task is spent trying to understand what the System Under Test (SUT) should be doing? (Not just is doing.) Once this information is understood, how is it preserved for the next engineer, the next release, or change order? If you are lucky it is in the test plan, but more typically buried in a test script or just lost, waiting to be rediscovered. By constructing a model of a system that defines the systems desired behavior for specified inputs to it, a team now has a mechanism for a structured analysis of the system. Scenarios are described as a sequence of actions to the system [as it is defined in the model], with the correct responses of the system also being specified. Test coverage is understood and test plans are developed in the context of the SUT, the resources available and the coverage that can be delivered. The largest benefit is in reuse; all of this work is not lost. The next test cycle can start where this one left off. If the product has new features, they can be incrementally added to the model; if the quality must be improved, the model can be improved and the tests expanded; if there are new people on the team, they can quickly come up to speed by reviewing the model.

2. Specifications

This paper assumes that the software to be tested is at the integration or system test phase of the development process (see Figure 1). The implied test objective is to insure that the system meets its requirements from the external point of view. The focus will not be on the implementation, but on how its users will evaluate it. The tests will measure the overall functional compliance of the system to the specification rather than code coverage. These tests are sometimes known as acceptance tests and are a traditional implementation of "black box" testing.

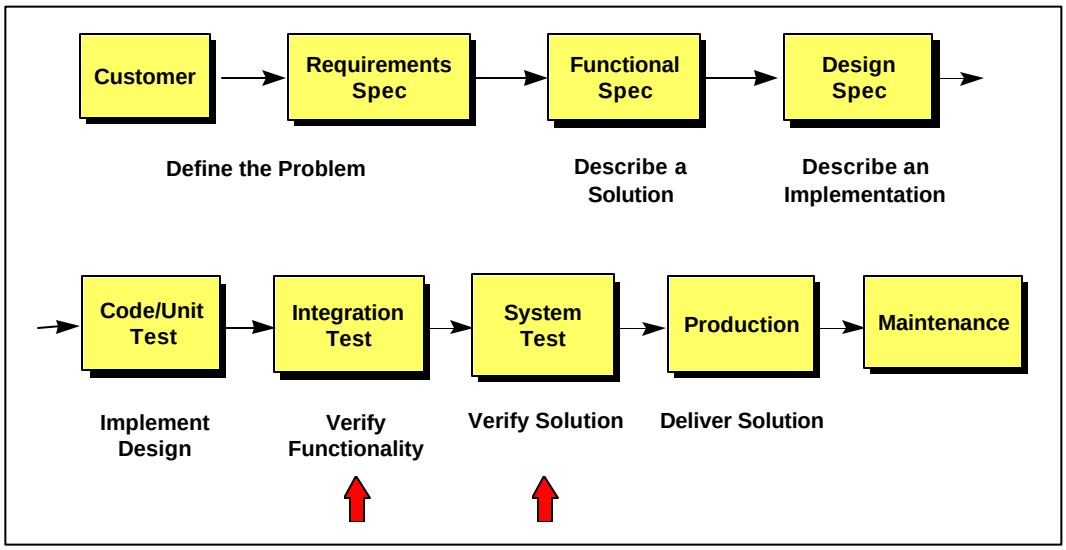

Figure 1. Although testing occurs throughout the development process, we will focus on the integration and systems test phases.

The first obstacle to overcome in developing tests is to determine the test target. While this may sound trivial, it is often the first place things go wrong. A description of the product or application to be tested is essential. The form the description can come in may vary from a set of call flow graphs for a voice mail system, to the user guide for a billing system's GUI. A defined set of features and / or behaviors of a product is needed in order to define the scope of the work (both development and test). The traditional means of specifying the correct system behavior is with English prose in the form of a Requirement Specification or Functional Specification [1]. The specification, when in prose, is often incomplete - only the typical or ideal use of the feature(s) is defined, not all of the possible actions or use scenarios. This incomplete description forces the test engineer to wait until the system is delivered so that the entire context of the feature is known. When the complete context is understood, tests can be developed that will verify all of the possible remaining scenarios. Another problem with textual descriptions is that they are ambiguous, (i.e., "if an invalid digit is entered, it shall be handled appropriately.") The 'appropriate' action is never defined; rather, it is left to the reader's interpretation. For example, you think we should allow the user to retry the entry, but I think we should abort the command. If you wrote the code and I wrote the tests, they would fail and we will waste time resolving an issue that could have been done before we even started. At a recent Requirements Engineering Conference, papers and presentations described the root cause of 60-80% of all defects as incorrect requirements specifications. [2]

Applying a model at this level in a development process can dramatically reduce the ambiguity - and hence, errors. The organization does not have to be at SEI Level 3 or 4 [3] to use these techniques; they can be applied anywhere. The reality is that modeling does not represent a new skill. Test engineers always build models. The only question is whether or not the models are in a persistent form. The 'model' may only exist for a short time and live on a napkin or in the mind of the engineer. In order to write a test script or test plan, an engineer must understand the basic steps required to use the system. Modeling at the behavioral level is very similar to flowcharting; the major transactions in the use of the product are defined in a graphical format. The sequence of actions that could occur during the use of the system are defined. The actions that "could" occur also imply that that there may be more than one possible action at a specific point in the process. Most modeling techniques support the idea that there are multiple possible "next" actions. Many methodologies are based on the concept of a state machine (see Figure 2), where the transactions represented by arrows in the diagram correspond to the actions, while icons in a variety of shapes represent the states. Some modeling techniques support hierarchical models, where a state can be replaced by a 'call' to another model which defines the behavior within the state. Hierarchical models allow complex behavior to be decomposed into simpler lower level models. Additional modeling capabilities include the use of conditionals, or predicates, to make the transactions (arrows) dependent on variables or the current system context. Other textual formats and notations have been employed in both research and industry include, SDL [4], and Z [5] both common to the communications sector. Less formal but no less appropriate is the notations used

by Beizer in his book entitled "Black Box Testing" [6] which reviews several different types of applications and uses a simple textual notation to define a model appropriate for each application.

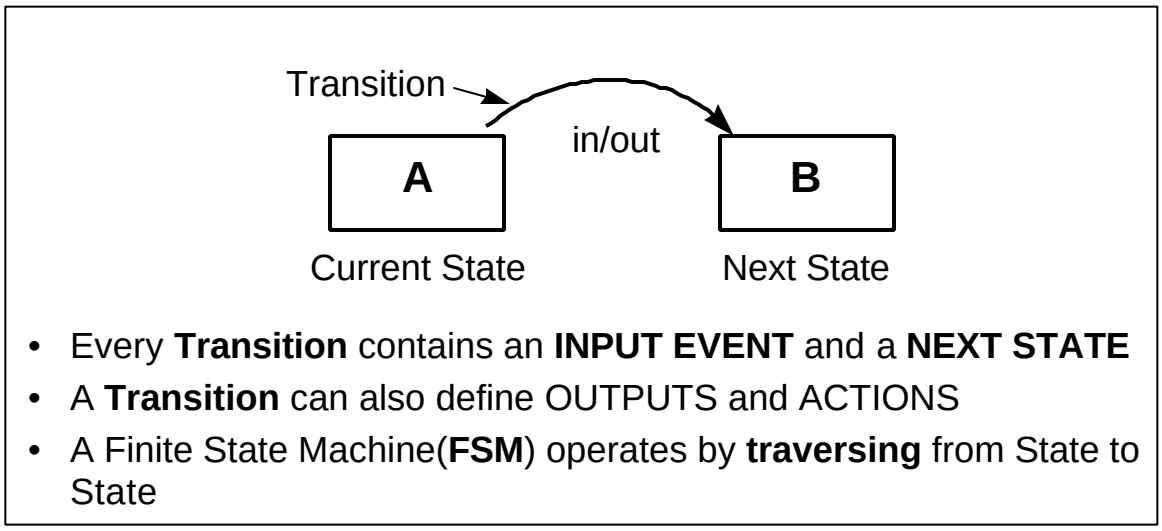

Figure 2. A Finite State Machine is Composed of States and Transitions

Developing a specification in the form of a model, even if done late in the process, is a very effective means of; 1) discovering defects in the system (many are made visible by the modeling effort alone), 2) rapidly defining the basis for use scenarios of the system, and 3) preserving this investment for future releases or other similar systems. Furthermore the process of developing a model can take place in a measured series of small incremental steps

3. Designing Tests

Subsequent to construction of a model - either complete or partial - the issue of test generation can be addressed. The test objective is to verify that the system will behave properly when a sequence of user actions occurs. For the purposes of this paper, a test script is defined as the entire sequence of actions required to create a complete user scenario for the system - from start-up through all of the actions and ending with shut-down. A test script can be decomposed into a series of individual test primitives that accomplish specific actions. The primitives will be combined in a specific sequence to provide a test script that verifies a unique use scenario for the product. Test primitives will typically fall into one of the following categories:

1) Provide a stimulus to the system. This is the most obvious - it controls movement from one state to another in the behavioral models. It can be a user action like selecting a button on a GUI, invoking a function of an API, or dialing a number on a telephone. The action can be directly controlled from the test execution environment.

2) Verify that the system responded correctly. Verification can be difficult to accomplish because the system's response must be determined and then compared to an expected response and/or verify that we are in the correct 'state' in our system. The degree that verification is used and the means for performing verification will vary widely with application type and test objectives. Some examples of verification are: comparing the text in a window on a GUI, checking that a dialed phone rings, comparing the return value of a function, or establishing that a billing record is generated after a call. Not all actions require direct verification - sometimes the fact that the next stimulus will be accepted is an acceptable means of verifying that the system is currently in the correct state.
3) Set-up the system testing environment. Tests will often need to control the environment so that the next action will follow a predictable path. In many environments there are situations that have several potential outcomes from the same input. These are often due to multiple resources being available for the action. To make this situation deterministic, and therefore easier to test, the environment can be temporarily pre-allocated or constrained in order to force a specific sequence or response to occur. Examples of this include: making a phone line busy so that the call forwarding feature on it can be tested and changing the status on an accounting record so that an error condition can be tested.
4) Report and/or log the results. Depending on the degree of automation in the test execution environment commands can be embedded that will report the test results to a reporting system. These embedded commands range from simple print statements for log files, all the way to sophisticated inter-process communications with a test management system. In any situation, the means to record and analyze results should be planned.

There are several approaches that can be used to develop tests from a model [7-12]. Central to most of these is the concept of a path. A path is a sequence of events or actions that traverse through the model defining an actual use scenario of the system. Each element in a path, a transition or state, can have some test primitives associated with it. The primitives will define what test actions are required to move the system from its current state to the next state, verify that the state is reached or check that the system has responded properly to previous inputs. Once a path through the model has been defined, a test script can be created for that path. When this script is applied to the actual system, the actual system will follow the same sequence (or path) as defined by the model path from which the test script was extracted. This process can then be repeated for another path, which defines another use scenario, and verifies another sequence of actions. Many methods can be used to select paths, each with its own distinct objectives and advantages. Operational profiles, reliability/criticality data, and switch coverage all provide different tradeoffs to the type of tests and the resulting coverage.

Other criteria include the quality or thoroughness desired in the tests. The test objective might range from the need for a set of tests to verify basic functionality all the way to a complete product verification suite.

4. Case Study: Long Distance Calling Service - Hierarchical Call Flow

A long distance service company provides low rates and flexible call billing to their customers. The customer receives a discount rate while charging their calls to any of a number of accounts. When placing a call, the customer dials an access code before the called number, and is then presented with a series of prompts to determine the method of billing for the call. The customer may repeat the process, with a different billing method if desired. The service has a relatively simple basic call flow that can modeled as shown in Figure 3.

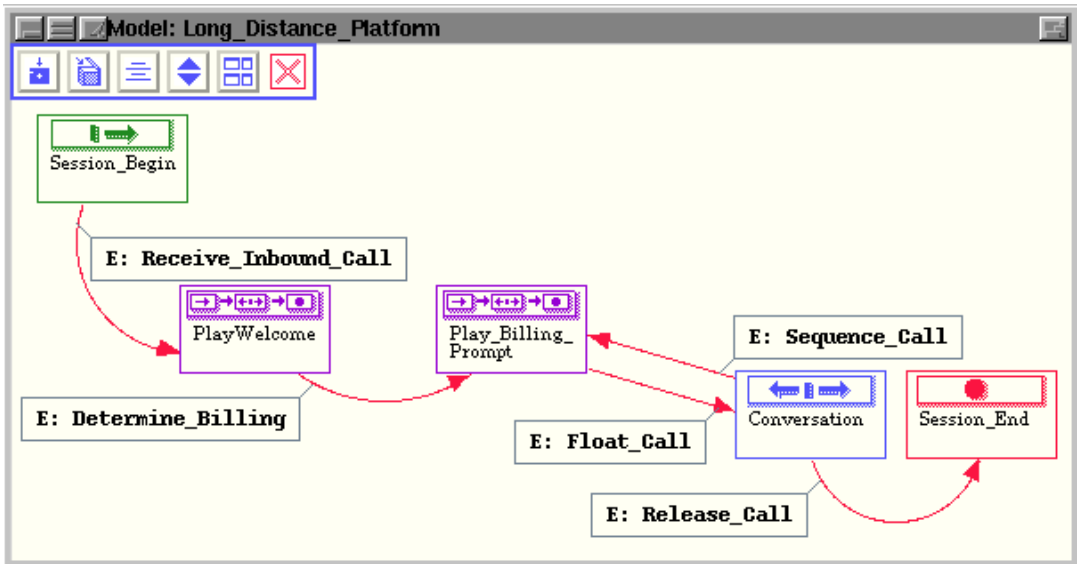

Figure 3. High Level Model of Long Distance Platform Call Flow

Modeling the Call Flow

The analysis required to build the model, is the same analysis necessary to design a test strategy. The requirements and assumptions of the platform must be well understood as well as the system response to various inputs. The model is simply a graphical representation of the behavior of the system. Once the system behavior is understood and captured in the model, the model can be used to generate tests to verify that behavior. In order to verify the long distance application platform, we need to identify the two call flow sequences defined by the model:

1. Inbound Call, Determine Billing Information, Float Call, Release Call
2. Inbound Call, Determine Billing, Float Call, Sequence Call, Determine Billing, Float Call, Release Call

The long distance service provider places three requirements on the application, the ability to bill to: 1) telephone calling card; 2) commercial credit cards, and 3) collect from the called party.

Adding Detail to the Model

The application model must be extended to include the behavior represented by these additional requirements. Rather than expanding the high level model to show the options presented at the Play_Billing_Prompt, we create a new level of hierarchy to represent the billing requirements. The hierarchical method provides a framework that allows the high level model to represent the basic call flow, while the lower level provides the detail necessary to accurately model the application's behavior. The framework is extensible to allow for additional levels of hierarchy when modeling complex call processing systems. The sub-model representing the billing options presented in the Play_Billing_Prompt model is shown in Figure 4:

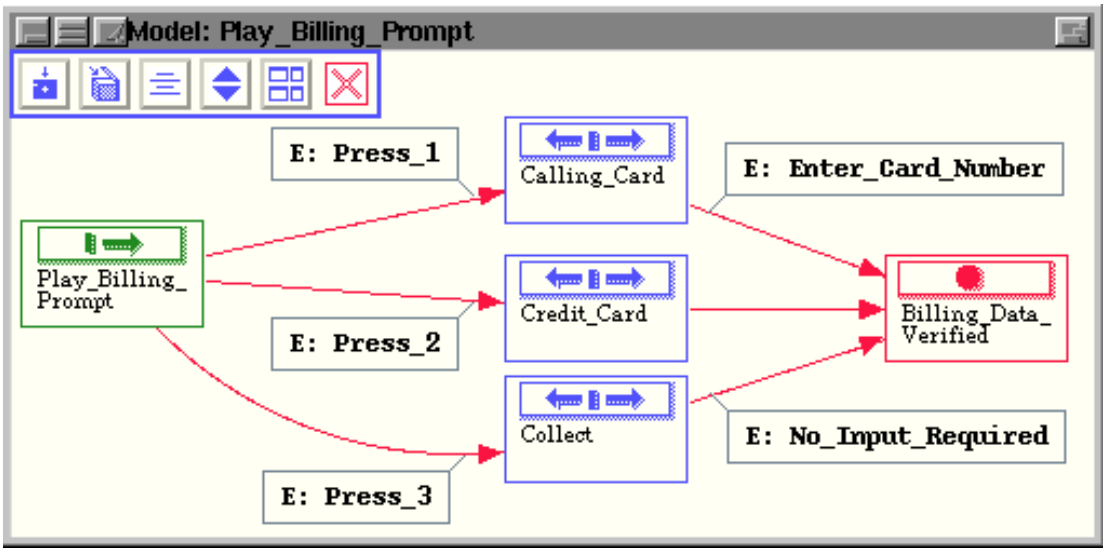

Figure 4. Detailed Sub-Model of Billing Options Prompts

Generating Paths from the Model

New features or requirements added to a system make the number of potential test sequences required grow dramatically. These three billing options increase the potential number of unique call-flow sequences by a factor of six, due to the ability to choose different billing methods while using sequence calling. As features are added to the platform, the complexity increases. The ability to manage the complexity of the system as new requirements are added is essential to platform verification. Manually generating the twelve sequences described by this system would be tedious and much of the call-flow development and analysis would have to be repeated as new requirements are placed on the system. The model provides a reuse framework for further development

The model also provides a compact representation of the system requirements that are used to automatically generate the call flow sequences. Automated test generation algorithms can be applied to the model to determine the call-flow sequences and coverage necessary for platform verification. As new requirements are added to handle additional billing options - international calling plans and account

queries - the application complexity will increase by orders of magnitude. By developing a model that captures the system requirements, those responsible for platform verification can manage the explosive complexity growth over time by adding hierarchy and incremental detail to a model. This is much more efficient than manually inspecting the system requirements and manually generating call flow sequences for test cases after each change. The model can be reused through the product life cycle or for other similar products on the platform. In a typical application of this approach, test engineer productivity has increased by a factor of five to ten over conventional manual approaches (see Figure 5).

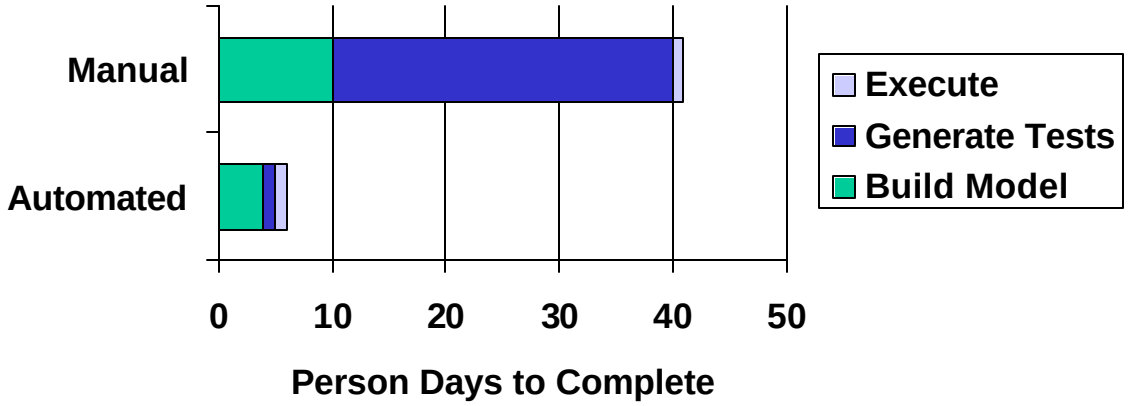

Figure 5. Tests for Call Waiting Feature was completed in 12% of the time

5. Case Study: Workorder System-Modeling the Entire Test Environment

A software supplier provides a workorder management and tracking system for operating companies. The system consists of User Interface (UI) applications linked together by a workflow application and a database management system (DBMS). The purpose of the modeling outlined in this study is to automate field level verification of the forms based UI, DBMS and basic transaction flow support in the workflow application. The goal of the modeling project was to develop a single model of this application environment. The testing environment consists of three different testing components, each with a unique set of requirements for the test generation process. A single model is used to produce :

- Scripts for functional tests through the UI
- Workorder records for verification of transaction support by the workflow application
- Test management system header information and build process for each test

Modeling the Workorder Application

The workorder application is specified by a Feature Requirements Specification, a set of operational scenarios and process flows, and database definitions. All of these sources are used to derive the information necessary to build a model. A forms-based system can be described as a series of transactions in a model; States, Transitions and Events are used to represent the behavior of the system.

Data structures within the model are used to represent the database record of the workorder, and conditional instructions are used to model the transaction semantics (i.e. dependencies) of the application. The model is built from the perspective of a user of the system. All of the actions that a user can invoke are represented graphically via arrows in the model.

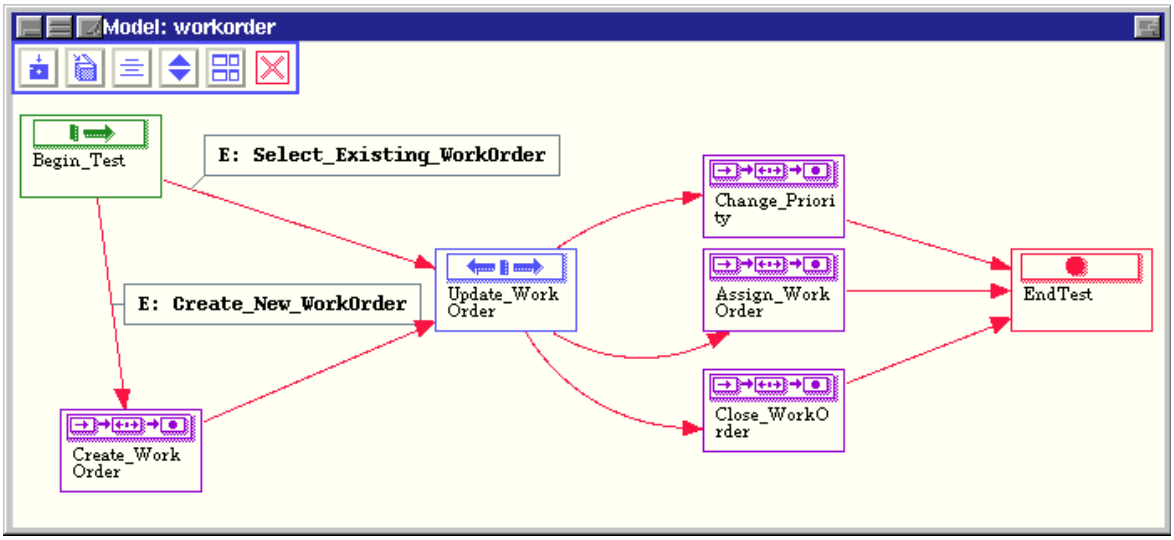

Figure 6. Model of Workorder Application

Figure 6 depicts a model that represents the initial processing of a workorder by a user of the application. The feature specification defines that the user must either "select an existing workorder" or "create a new workorder" before accessing the forms to submit a "change", "assign", or "close" operation. This is shown in the model as separate transitions (arrows) leaving the "Entry" state. One transition represents the event "Select Existing WorkOrder" the other "Create a New WorkOrder". This simple graphical model captures the flow of the system by representing the state of a workorder as icons and valid operations in that state as transitions. The model also represents the behavior of the forms UI application built on top of the transaction processing system. From this behavioral description three types of test components are developed: a set of test scripts that functionally exercise the system, database records used to verify the correct behavior, and test management information to report the results.

Modeling Behavior to Generate Functional Tests Through the User Interface

The Functional Requirements Specification and process flow documents define the behavior of the Application Under Test (AUT). The AUT in this case is a forms-based system that builds a record and submits a transaction. The documents contain functional descriptions of the application as well as detailed data descriptions. The specifications are used at the highest level to determine valid use scenarios for processing the forms. At the lowest level they determine valid inputs and expected outputs for each field in a given transaction as well as error message descriptions. The model based approach captures this detail as well has the behavior represented by the specifications.

The model is processed in order to produce the executable test files required by the Test Environment. The workorder application is very straightforward to test. A simple scripting language is used to traverse from field to field and form to form. Elements of the scripting language are embedded on each transition in the model in such a way that when the model is processed, the model defines a test of the behavior of the application. For each valid flow (or path) derived from the model, a new test is created. These paths include the operational scenarios specified in the requirements specification as well as other valuable test scenarios.

Modeling Data and Maintaining Application Context within the Model

The model can be used to describe flows, but in order to represent the specification and build meaningful tests, data must be represented as well. Within the model variables are defined whose values can be changed, conditionally checked and output at any point in the flow. The model uses these variables to both maintain context and generate data in the test scripts. Data values are conditionally checked within the model to determine which paths are appropriate in the current context. Multiple values of data are modeled to verify the application behavior over specific domains. For example, a variable called "Transaction" is maintained to represent the state of the "current transaction". When the model is being processed and enters the "Assign_WorkOrder" state in Figure 6, the value of "Transaction" will be set to "ASSIGN". This action defines the current context for later use in determining appropriate paths. The string is also used to define input values in the forms, and later verify the DBMS record. The data used to test the application requires more than modeling the values of data. The data fields in a workorder are defined in terms of:

- the transaction for which they are valid, or they are common to all transactions
- necessity, whether they are required or optional
- type, input or output
- data type, length and value

Figure 7 shows the model of the forms system for creating a workorder. When preparing a workorder for the CREATE transaction we have separated the fields of the form into three sub-groups. The fields common to all forms are filled in first (CommonFields), then the fields "required" for CREATE are filled in (CreateReqField), then the "optional" fields are selectively filled in (CreateOptField). Each group is further described in a sub-model, decomposing the system via hierarchy. The sub-models represent individual fields in the form; they contain the detail necessary to generate input data of valid and invalid type, length and value. Optional fields are represented by branched transitions (arrows) with conditional expressions embedded in the transition. An example of a conditional transition occurs in Figure 7, "Transaction==CREATE" (this only allows transactions of type CREATE through this sub-model).

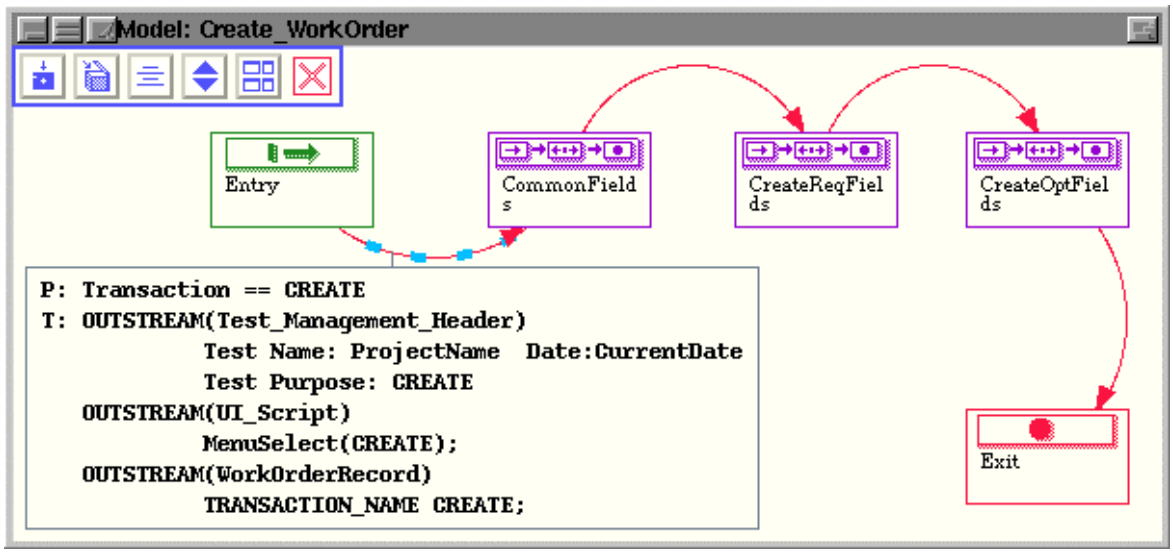

Figure 7. High Level Model of the "CREATE" Form

Modeling Transaction Flow to Generate Database Verification Records

The Workorder Application is used to create and edit workorder records and submit transactions. The model captures the workorder process flow, and the values of data throughout this flow. Driving the workorder system in a test requires that the data values be output to a forms processing script in the proper sequence. Verification of the transaction processing is more difficult. No feedback is provided through the UI as to the success of the transaction. The scripting facility has no on-screen verification, so reading back the record in a form is not possible. The proper function of the workflow application is verified by querying the database directly with a separate database utility. When a path through the model creates a workorder record the verification record is also created as part of the process. For example, if the current state is CREATED, and an UPDATE transaction is submitted, the model contains enough information about the behavior of the system to generate two workorder records; a test record is generated for the forms UI, and a verification record for the DBMS. A query is used to confirm that the output record matches the verification record, and if it does not, an error is logged.

Using the Model to Generate Test Management System Files

The test process in this company requires all tests to be archived, logged and run from a central test repository. Each test is stored in a test management system and contains an identifier and test header. The information in the test header is of two classes, basic template information (test name, version number, current date) and test purpose. The purpose of the test is a description of the behavior executed by the test script. Both of these pieces of information are readily available in the model. The model automatically generates the test management system file for each path generated by the model.

Summary

In a workorder system, the forms system definition and the transaction flow parallel each other. A single model is used to represent them both as a single flow with parallel output streams. The model can be designed to accommodate any output of the system, even descriptive text for a test management system. By building a model that includes all three major components of the test execution environment we have automated the entire testing process, from test case specification to the logging of results. Using a traditional manual approach these steps would be done sequentially, at a much greater cost. With the development of a few script files, over 1000 tests were automatically generated, registered in the test management database, remotely executed and verified while successfully maintaining a results file. A major improvement in both the efficiency of the team and quality of the product was achieved (see Figure 8).

	Manual Approach	Models & Automation
Build Model	-	10 days
Typ. Change	5 days	15 min
Test Gen	20 min/test	3 sec/test
Test Exec	3-10 min/test	2 min/test
Report Gen	3 min/test	0*

* Automation included Test Management (Buster) integration

Figure 8. Process Metric Comparison for Workflow based system

6. CONCLUSION

To make this a practical solution the economics must be understood. If the technique does not save money, it will not be used. The traditional metrics used to justify a purchase of software or to warrant a change to an existing process include: lower cost, increased quality, or reduced time to market. Model Based Testing is a methodology that has proven its ability to provide dramatic improvements in all three of the metrics. The overriding challenge to its more broad based acceptance is education. We hope we have helped in addressing that problem.

Acknowledgments
We would like to acknowledge and thank the teams at Lucent Holmdel and Liberty Corner for their help in developing some of the models and systems described here. The teams include: Dr. Sadik Esmelioglu, SanSan Ting, Chrysanthi Kefala, Debby Kau and Linda Cheng.

References:

[1] IEEE standard for Requirements Specification (IEEE/ANSI Std. 830-1984) , IEEE Computer Society, (830-1993) IEEE Recommended Practice for Software Requirements Specifications (ANSI), IEEE Standard for Software Unit Testing (ANSI), IEEE Standard for Software Verification and Validation Plans (ANSI) found at: http://standards.ieee.org/catalog/it.html.

[2] Proceedings of the Third IEEE International Symposium on Requirements Engineering, IEEE Computer Society, 1997.

[3] Paulk, M., Curtis, B., Chrissis, M.B., and Weber, C., Capability Maturity Model, Version 1.1 The Software Engineering Institute, Carnegie Mellon University. Found at: http://www.sei.cmu.edu/products/publications/96.reports/96.ar.cmm.v1.1.html

[4] ITU-T. ITU-T Recommendation Z.100: Specification and Description Language (SDL). ITU-T, Geneva, 1988. More can be found at http://www.sdl-forum.org/.

[5] Spivey, M., The Z Notation: A Reference Manual, Second Edition. Prentice-Hall International, 1992.

[6] Beizer, B., *Black Box Testing*, New York, John Wiley & Sons, 1995. ISBN 0-471-12094-4.

[7] Fujiwara, S., Bochmann, G., Khendek, F., Amalou, M., and Ghedamsi, A., "Test Selection Based on Finite State Models", *IEEE Transactions on Software Engineering*, Vol. 17, No. 6., June 1991.

[8] Chow, T. S., "Testing Software Design Modeled by Finite-State Machines," *IEEE Transactions on Software Engineering*, Vol. SE-4, No. 3, May 1978.

[9] Holzmann, G., *Design and Validation of Computer Protocols*, AT&T Bell Laboratories, Prentice Hall, 1991.

[10] Musa, J.D., "Operational profiles in software reliability engineering," *IEEE Software*, 10(2), pp 14-32.

[11] Apfelbaum, L., "Automated Functional Test Generation", Proceedings of the Autotestcon '95 Conference, IEEE, 1995.

[12] Savage, P., Walters, S, and Stephenson, M., "Automated Test Methodology for Operational Flight Programs", Proceedings of the 1997 IEEE Aerospace Conference, 1997.