Annals of Software Engineering 3 (1997) 291–317

Software requirements and acceptance testing

Pei Hsia and David Kung

Department of Computer Science and Engineering, University of Texas at Arlington, Box 19015, Arlington, Texas 76019–0015, USA
E-mail: {hsia, kung}@cse.uta.edu

Chris Sell

Lockheed/Martin, Fort Worth, TX 76101, USA

In this paper we 1) review industry acceptance testing practices and 2) present a systematic approach to scenario analysis and its application to acceptance testing with the aim to improve the current practice. It summarizes the existing practice into categories and identifies the serious weakness. Then, a new approach based on the formal scenario analysis is presented. It is systematic, and easily applicable to any software or system. A simple, yet realistic example is used to illustrate its effectiveness. Finally, its benefits and its applicability are summarized.

1. Introduction

Acceptance testing consists of comparing a software system to its initial requirements and to the current needs of its end-users or, in the case of a contracted program, to the original contract [Meyers 1979]. It is a crucial step that decides the fate of a software system. Its visible outcome provides an important quality indication for the customer to determine whether to accept or reject the software product. It is, therefore, categorically different from other types of testing where the intent is principally to reveal errors.

Acceptance testing is focused on major functional and performance requirements, man-machine interactions, specified system constraints, and the system's external interfaces. The major guide for acceptance testing is the system requirements document and the primary focus is on usability and reliability [Hetzel 1984]. User involvement, therefore, is crucial. In brief, the goal in acceptance testing is to:

(1) verify the man-machine interactions,

(2) validate the required functionality of the system,

(3) verify that the system operates within the specified constraints, and

(4) check the system's external interfaces.

Presently there is no industry-wide standard for acceptance testing and, as such, the practice is quite ad hoc and varies greatly. There is no systematic method available to help testers construct, formalize, and verify acceptance testing models, and use them to guide and direct the process. The problems with an ad hoc approach are:

(1) the test plan generation relies heavily on the understanding of the software system and its application domain,

(2) there are no formal test models representing the complete external behavior of the system from users' perspective,

(3) there is no methodology to generate all possible usage patterns necessary to test the external behavior of the system,

(4) there is a lack of rigorous acceptance criteria, and

(5) there are no techniques for verifying the correctness, consistency, and completeness of the test cases [Hsia *et al*. 1994a].

Acceptance test cases are typically generated manually by a group of domain experts via study of the requirements document with a view toward determining whether the software system satisfies the requirements. The generated tests are classified and collected into a test suite. Unfortunately, there is no way to rigorously demonstrate that the test suite covers all the requirements (completeness). Furthermore, there is no way to indicate that the requirements cover the entire test suite (correctness). Finally, there is no technique to demonstrate that there are no inconsistent test cases (consistency). The problems of completeness, consistency and correctness of a test suite are difficult to address because there is no effective test criteria and no fundamental acceptance test model. The relationship between a set of requirements and a test suite is many-to-many and, sadly, it is nearly impossible to automate.

Software requirements are written in English prose, with some charts and/or graphs for clarification at best. They are the sole basis for acceptance of the software product. Two questions need to be asked:

(1) can we formulate a systematic process that can lead one from a set of requirements to a proper set of test cases for acceptance testing and

(2) can we establish a new way to write software requirements that will significantly support the acceptance testing component of a software system?

Researchers have shown that using formal specification languages in the requirements phase can reduce the effort in acceptance testing because it helps one to generate test cases automatically [Ferguson and Korel 1996]. However, formalisms are often used at the expense of communication between customers and developers, since most customers are not well versed in formal specification languages. This approach may be useful and effective in certain domain specific applications as it can facilitate the automation of many activities required in the software development process. However, the prospect of popularizing this approach is probably not very bright.

We are more interested in addressing the first question. We accept the existing way of expressing software requirements and put forth a new technique called scenario analysis to facilitate the derivation and generation of test cases from requirements. Using this technique, a usage model is obtained which serves as a foundation and context for discussing the completeness, consistency and correctness of acceptance test suites.

This paper has two objectives:

(1) to review industry acceptance testing practices and

(2) to present a systematic approach to scenario analysis and its application to acceptance testing with the intent to improve the current practice effectively.

The remainder of the paper is organized as follows: section 2 discusses the current state of acceptance testing, providing a taxonomy of the methods both used and suggested. Section 3 gives a description of scenario analysis and section 4 shows how it can be used for acceptance testing as illustrated by a realistic example. Section 5 discusses the benefits of scenario analysis and its applicability while section 6 provides concluding remarks.

2. State-of-practice of acceptance testing

The practice in industry is characterized by the following steps:

(1) study the requirements document,

(2) obtain help from a domain expert,

(3) develop a set of test cases, and

(4) demonstrate to the customer, by using the test cases, that the software indeed possesses the required functionality as formally or informally specified.

There are many problems with this process. The first three steps all involve sharp subjectivity. There is no common frame of reference within which to interpret requirements, the degree and extent of expertise of domain experts vary greatly, and the ways to develop test cases are often problem dependent. No two different teams will develop the same set of test cases for acceptance testing given the same information. Many experiences have been published, and some of the proposed approaches are included for reference.

A taxonomy of acceptance testing is developed from the current literature on testing. Four categories emerged from the survey which include

- traceability methods [Davis 1993; Deutsch 1982; Kleinstein 1988; Krause and Diamant 1978; McCabe 1983; Samson 1990; Wallace and Cherniavsky 1986; Wallace 1990; Wardle 1991],

- formal methods [Bauer *et al.* 1978; Bauer and Finger 1979; Chandrasekharan *et al.* 1989; Chow 1977; Dasarathy and Chandrasekharan 1982; Ferguson and Korel 1996; Davis 1980; Hsia *et al.* 1994c; Huebner 1979; Ramamoorthy *et al.* 1976; Worrest 1982; Alford 1977; Davis and Ranscher 1978],
- prototyping methods [Lea *et al.* 1991; Gomaa and Scott 1981; Andriole 1989], and
- other methods [Compton *et al.* 1977; Dunn and Ullman 1982; NBS 1984; Janowiak 1990; Linehan 1988; McCabe and Schulmeyer 1985; Musa and Ackerman 1989; Osder and Decker 1982; Protzel 1988; Quirk 1985; Ray 1989; Tausworthe 1979; Lipow and Thayer 1977].

Traceability methods work by creating a cross reference between requirements and acceptance test cases, fabricating relationships between test cases and requirements. They allow testers to easily check the requirements not yet tested. They also show the customer which requirement is being exercised by which test case(s). It is a straightforward way to match the test cases to the requirements. However, each cross reference is created by the tester subjectively.

Formal methods apply mathematical modeling to test case generation. They include the application of formal languages, finite state machines, various graphing methods, and others to facilitate the generation of test cases for acceptance testing. A Requirements Language Processor (RLP) is a good example. It produces a finite-state machine (FSM) model whose external behavior matches that of the specified system. Tests generated from the FSM are then used to test the actual system.

Prototyping methods begin by developing a prototype of the target system. This prototype is then evaluated to see if it satisfies the requirements. Once the prototype does satisfy the requirements, the outputs from the prototype become the expected outputs from the final system. Therefore, prototypes are used as test oracles for acceptance testing.

Other methods include statistical methods, structured analysis (SA) and simulation. Statistical methods use software-reliability measurement or base acceptance on empirical data about the behavior of internal states of the program. Software-reliability measurement is the probability of failure-free operation relative to a theoretically sound and validated statistical model. Structured analysis (SA) methods involve the use of SA techniques (entity-relationship diagrams, data flow diagrams, state transition diagrams, etc.) in aiding the generation of test plans.

Simulation is more a tool for acceptance testing than a stand-alone method since it is not used in the development of test cases. It is used for testing real-time systems and for systems where "real" tests are not practical. The following subsections summarize these methods in detail.

2.1. *Traceability methods*

Incorporating traceability into the requirements document is one of the frequently discussed methods for acceptance testing. A software requirements specification (SRS) is traceable if it is written in a manner that facilitates the referencing of

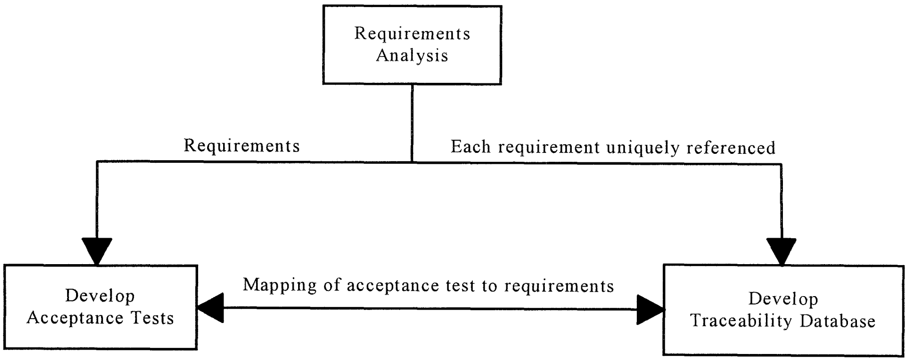

Figure 1. Traceability methodology.

each individual item of requirements stated therein. This can be done by numbering every paragraph in an SRS hierarchically and then including only one requirement per paragraph or referencing each requirement by paragraph and sentence number. This can also be done by numbering every requirement in parentheses immediately after the requirement appears in the SRS or by using a convention for indicating requirements; e.g., always use the word "shall" in the sentence containing the requirement. This is sometimes referred to as requirements tagging.

Once the SRS has been made traceable, a database can then be used to help automate traceability tasks. This database holds a matrix, or checklist, which ensures that there is a test covering each requirement. It also keeps track of when the tests have been completed (or failed).

Figure 1 shows how acceptance test cases and the traceability database are developed after the requirements analysis phase is complete. The database can be developed in tandem with the test cases or as a separate step after the test cases are finished. We sample several approaches, contrasting their features, to illustrate the spectrum approaches in this category. Our samples include Test Evaluation Matrix (TEM) [Krause and Diamant 1978], REQUEST [Wardle 1991], and SVD [Deutsch 1982].

The TEM shows traceability from requirements to design to test procedures allocated to verify the requirements. It first allocates requirements to three test levels. Requirements testing is then performed by selecting test scenarios that satisfy test objectives, represent test missions, and exercise requirements allocated to that level of testing. Test cases are derived from test scenarios by selecting the target configuration and user options necessary to guide the unit, subprocess, or process through the desired paths.

The REQUEST (REquirements QUery, and Specification Tool) tool has been developed to automate much of the routine information handling involved in providing traceability. It is a database tool consisting of the Contractual Requirements Database, a Test Requirements Database and a Verification Cross Reference Index (VCRI). First, the Contractual Requirements database is built by identifying the individual requirements. Then, a Test Requirements database is created. REQUEST is used to generate a test specification document from this test database. With additional user input such as the test level and the test method to be used, REQUEST then generates the VCRI. REQUEST ensures that there is a test covering each requirement and it is the VCRI that provides full forward and reverse traceability between a requirement, its test, and its design.

The SVD (System Verification Diagram) tool also connects test procedures with requirements. The SVD contains threads which are stimulus/response elements that represent an identifiable function or subfunction associated with a specific requirement. These threads represent the functions that must be tested and the paths through the SVD are the sequences of functions that are to be tested. The cumulation of all the threads forms the acceptance test. It is similar to the scenario analysis method presented in this paper. The major difference is that the development of the threads in SVD is done after the requirements analysis phase and the user is not involved in the development of the threads. Also, threads are less elaborate than scenarios and the testers exercise more subjectivity when using threads.

2.2. Formal methods

Another type of method being used for acceptance testing centers around the use of Requirement Language Processors (RLPs) and finite-state machines. An RLP produces a representation of a finite-state machine. This finite-state machine is a model whose external behavior matches that of the specified system. Generating tests for a finite-state machine could be done at a level of fidelity similar to the actual completed system. Work in this area also includes the development of Test Plan Generators (TPGs) [Bauer *et al*. 1978; Bauer and Finger 1979; Chandrasekharan *et al*. 1989; Dasarathy and Chandrasekharan 1982; Davis 1980] and Automatic Test Executors (ATEs) [Bauer *et al*. 1978; Bauer and Finger 1979; Davis 1980; Worrest 1982].

A TPG is an interactive tool that analyzes the functional description produced by a RLP and produces a set of executable test scripts. An ATE reads these test scripts, runs the tests, and produces a test report. As with the traceability method, the production of the test plan is a separate step from requirements analysis.

The use of a RLP has also been applied specifically to the validation of telephone switching systems [Huebner 1979]. Here, the feature descriptions produced by the RLP are processed by a System Test Complex (STC). The STC generates test case scenarios from the feature descriptions. The scenarios are then executed by the STC, which then produces a test report. With this method some of the work of generating the test plan is done in the requirements analysis phase because the scenarios come directly from the feature descriptions produced by the RLP. Figure 2 shows how the RLP, the TPG, and the ATE are coordinated to perform the testing function.

2.3. Prototyping methods

Using prototypes is another method being for acceptance testing [Beizer 1984; Davis 1990; Lea *et al*. 1991]. With this method a prototype is developed and evaluated

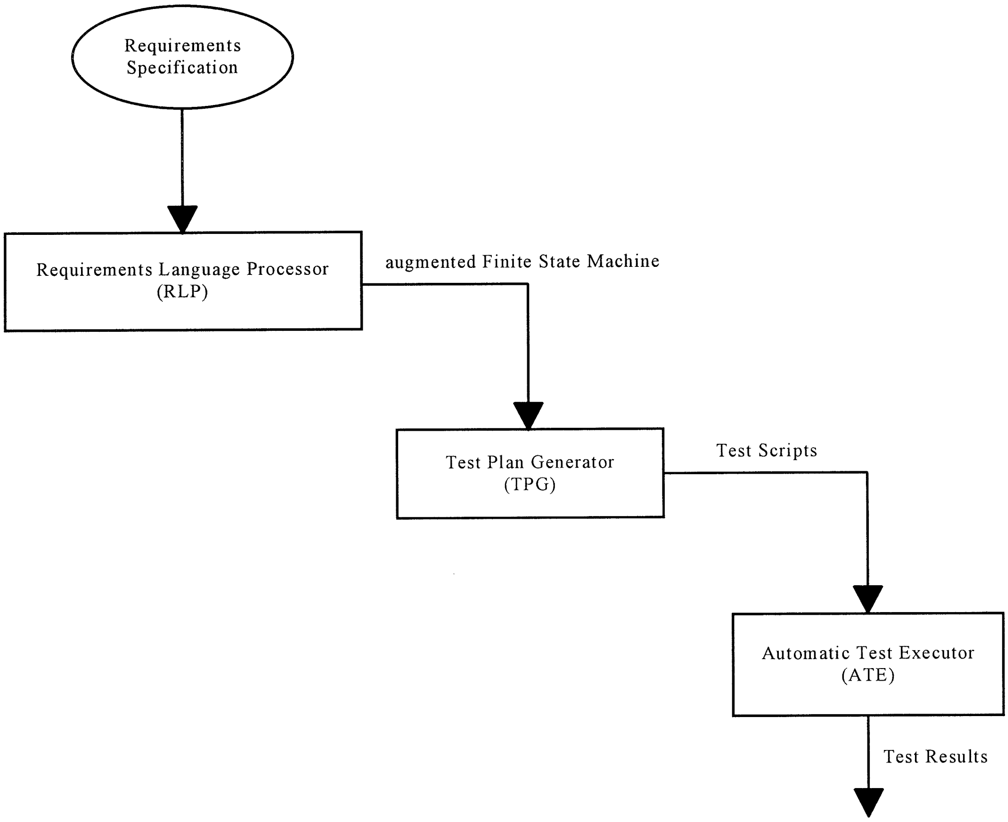

Figure 2. A finite state machine example method.

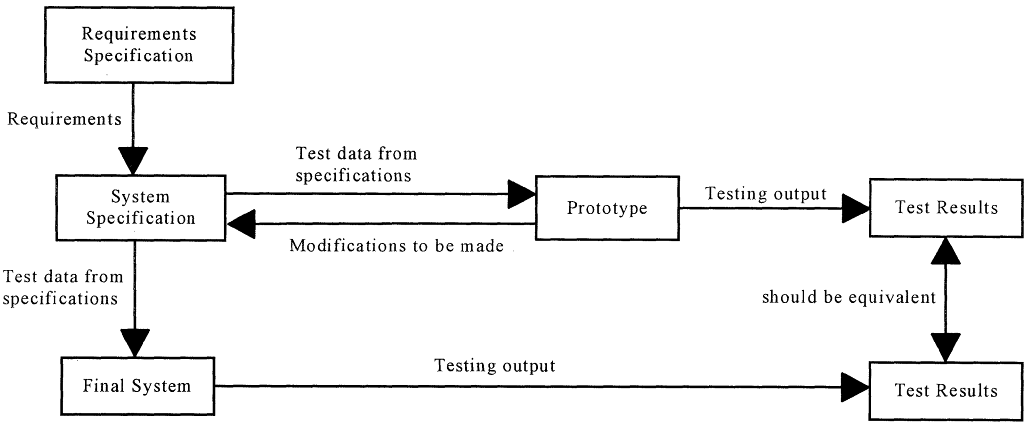

Figure 3. Prototype methodology.

to verify if it satisfies user requirements. If it does not, the requirement specifications are updated and a new prototype is built. Once the requirements are met, the corresponding specifications are used in the design and implementation stage of the target system. The prototypes are evaluated with input data generated from the specifications. These data are then used as input for acceptance testing of the final system (see Fig. 3). The outputs from the prototype evaluation are the expected output of the acceptance testing.

2.4. Other methods

This section covers other methods that were found but which are not widely used or have not gained a significant amount of coverage in the literature. Also included are papers that were written as general guidelines but which do not give a specific methodology for implementing those guidelines, and articles that discuss what is involved in performing an acceptance test for a specific application without necessarily giving a methodology for performing or developing the test [Janowiak 1990; Linehan 1988; Ray 1989].

2.4.1. Empirical data

One proposed method for acceptance testing is based solely on empirical data about the behavior of internal states of the program [Protzel 1988]. The process is carried out by defining test-bits within the application software and combining them into a pattern which reflects the internal state of the program. This process is similar to instrumenting a piece of hardware to determine how it is functioning internally while it is operating. Empirical data about this pattern is collected by an oracle which stores the values and the frequency of the test-bit patterns in the form of a distribution. This distribution contains information about the correct program behavior and can be used as an acceptance test for the program. The results are evaluated by comparing the pattern observed for each run to the data stored in the distribution. If the observed pattern is not in the distribution, one of two things may have happened. Either an error in the software has occurred or a correct but unusual event has occurred. In either case a warning message is produced.

2.4.2. Software-reliability measurement

Still another method focuses on the use of a software-reliability measurement which describes the probability of failure-free operation relative to a theoretically sound and experimentally validated statistical model [Musa and Ackerman 1989]. This method uses statistical analysis to determine the probability of failure (outputs not conforming to the requirements) per CPU hour. (This is sometimes referred to as mean time between failure or MTBF.) During system testing the use of reliability measurement presupposes that the definition of significant failures has been specified, an operational profile has been specified, and, for the defined failures and operational profile, a failure-intensity objective has been set to a desired level of confidence. For acceptance testing the software is verified by checking the failure intensity on an acceptance chart by making random test runs selected from the operational profile upon which the developer and supplier have agreed. An acceptance chart is a 2-D plot of failures vs. normalized failure time. A diagonal line from $(0, 0)$ to infinity is the threshold at which any point falling above causes the software to be rejected. The slope of this line is set according to the failure rate that is considered acceptable.

2.4.3. Structured analysis

Structured analysis (SA) can be applied to systems that already exist, have no requirements document, and which need an acceptance test defined for them (basically reverse engineering the system) [McCabe and Schulmeyer 1985]. The method used for system testing has been to draw a Data Flow Diagram (DFD) of each existing subsystem and to derive the system test flow from the DFD's. The DFD's provide a depiction of the control flow through the integrated system for thorough system acceptance testing. Of course, the use of SA and DFD's from the beginning of a project can aid in acceptance test generation.

2.4.4. Simulation

The use of simulation also has its place in acceptance testing [Compton *et al.* 1977; Osder and Decker 1982; Quirk 1985]. Quite often simulation is the nearest one can get to real life use for checking both the specification and the system in terms of the real requirements of the system. This is especially true when validating real-time systems and testing the actual system is too dangerous or when trying to validate system performance characteristics. Another advantage of simulation is that the simulator and the actual system are usually based on different development personnel or environments. Even if the system specification and the simulator are incorrect due to difficulties in understanding the real world, inconsistencies are less likely to coincide. Therefore, testing a simulator may offer enhanced reliability expectations analogous to the use of diverse design and implementation techniques. In the area of system functionality and interfaces, simulation can help by exposing the incompleteness and ambiguity of the requirements specification. By demonstrating the specified requirements to the customer or end user, there is a better chance of revealing missing or misunderstood requirements.

2.5. Summary

All of these methods have their advantages and disadvantages. The advantage to the traceability method is that it ensures there is a test case covering each requirement. The advantage to the FSM methods is that the use of a RLP ensures the requirements are complete, consistent, and unambiguous. Prototypes are advantageous when developing high risk or never previously attempted software applications.

The other methods have the advantage of working for specific applications. The main disadvantage of all the methods is that they typically leave the user out of the test development process. Tests are usually developed with a bias towards the test developers' viewpoint. In other words, the quality of the tests developed is not always of a high standard because too much subjectivity is involved. Also, the test development process itself is usually a separate task from the requirements analysis phase. Hence, it necessitates more time, personnel and money. RLP methods suffer because they require precise and complete requirements early in the software life cycle, which is rarely practical. With scenario analysis these drawbacks are overcome without forfeiting many of the aforementioned benefits.

3. Scenario analysis

A scenario is a description given by a user of how he/she will or wants to use a software system. It consists of a sequence of events and responses which mimic the interactions between a user and a system. The purpose of a scenario is to provide an explicit and concrete example of how some human activity could be supported by technology [Nardi 1992]. Scenario analysis then, as it relates to software systems, is the process of analyzing, understanding, and describing system behavior in terms of particular ways the system is expected to be used. The end product of scenario analysis is a document consisting of a set of sufficiently complete, consistent, correct, and validated scenarios. This document is used as a guide for design and testing, including user acceptance and system validation testing [Hsia *et al.* 1994a].

Much work has been done on applying the concept of scenario analysis. While this work is beneficial, most of it lacks a formal methodology that can be used to apply it. Hsia *et al.* [1994a] have created a formal method to systematically identify, generate, analyze, and verify a set of scenarios of a software system. An informal description is given below. Readers interested in the rigorous definitions are referred to [Hsia *et al.* 1994c].

An event is a specific stimulus to a system, as a result of which either the state of the system is changed and/or another event is triggered. An external event is one that occurs outside the system. Most events usually have a very short duration. An event type defines all the possible events that have similar attributes. A scenario consists of an ordered sequence of events which accomplishes a given functional requirement.

A user view is a set of scenarios as seen by a certain group of users. The user view describes the interactive behavior of the user with the system. An external interface view represents the behavior of an interface between a system and its interfaced external system. This interface represents the behavior in terms of inputs (stimuli), outputs (responses), triggered events, and reactive events. An external interface view consists of a set of scenarios as seen from the interface.

An external system view is a set of scenario schema which represent the external view of the system in terms of its user views and external interface views. An instance

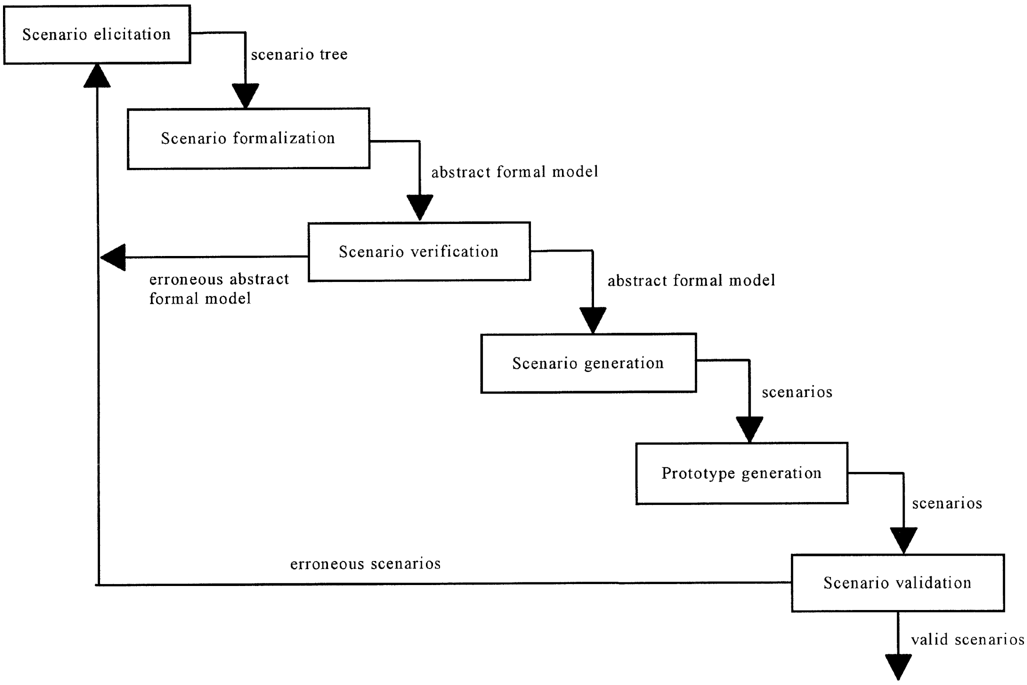

Figure 4. Scenario analysis model.

of an external system view consists of a set of concurrent, communicating scenarios in the system.

A scenario tree consists of a finite set of nodes N, a finite set of edges E, and a finite set of edge labels L. The node denoted as the root is the user-perceived initial state. The node set consists of a set of states as perceived by the user. For every edge between any two nodes, there is a label, which is an event type, associated with it. A label l of an edge between nodes $N1$, $N2$ shows that the state of the system changed from $N1$ to $N2$ due to the occurrence of the event type l. (See Fig. 5 for an example.)

The scenario-analysis process model involves six stages which are scenario elicitation, scenario formalization, scenario verification, scenario generation, prototype generation, and scenario validation (see Fig. 4). The following is a summary of formal scenario analysis. For a complete description as well as an extensive example, readers are referred to [Hsia *et al.* 1994a].

Scenario elicitation is conducted by the requirements analyst and the customer or the user together with a domain expert and is done to determine the required scenarios as seen from the users' point of view. The scenarios in a user's view are initiated by someone or something, either the external users, external stimuli, or functional components. This process begins by constructing a scenario tree (see Fig. 5) which is a mechanism that describes and represents all the scenarios in a particular user view. To construct a scenario tree for a particular user view, a root node is first built

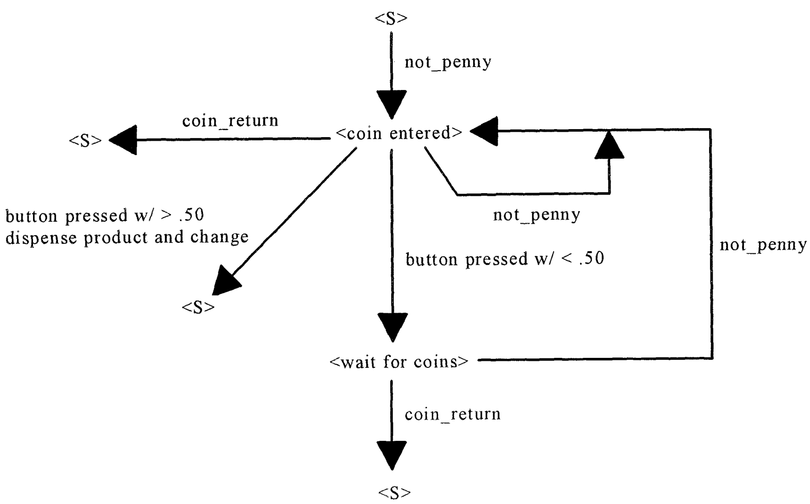

Figure 5. Scenario tree example for customer view.

to identify an initial system state. The analyst then queries the domain expert as to what event types can occur following this initial state. For each event type the user describes, a different branch (edge) is added from the root node. Each edge is labeled with the event type name. The node(s) at this first level now indicate the state of the system as perceived by the user after the occurrence of an event. The analyst repeats the process by adding nodes and branches until all possible sequences of events are identified. In the completed scenario tree, each path represents a scenario scheme for that user view.

As an example, consider the following simple set of requirements. Construct a vending machine that sells one product costing $0.50. Once $0.50 or more is deposited in the machine and the dispense product button is pressed, the product is dispensed along with change (if any). The machine will accept any coin except a penny. The machine also has a coin return function. The scenario tree for this is show in Fig. 5.

Scenario formalization is the process of converting each scenario tree into an equivalent regular grammar. This grammar is defined as a quadruple comprising of the states of the scenarios, the events, the starting state (also the ending state), and the set of rules governing the transitions from state to state. The formal definition of this grammar is given in [Hsia *et al*. 1994a] along with an algorithm to construct the grammar for a given scenario tree. This grammar is used to construct a conceptual state machine model that represents the system behavior from a particular user view. The nodes in this model are system states and the edges between the nodes are events. The grammar and the corresponding conceptual state machine comprise the abstract formal model. This model is used to represent and display the system behavior in terms of scenarios. The grammar for the tree in Fig. 5 is given in Fig. 6. The conceptual state machine for the customer view is given in Fig. 7.

Scenario verification is the process of verifying the abstract formal model to uncover inconsistencies, redundancies, and internal incompleteness in the scenarios.

Events = {not_penny, coin_return, button pressed w/ < .50
button pressed w/ > .50/dispense product and change}
States = {⟨coin entered⟩, ⟨wait for coins⟩}
Starting state = Customer
Rules = {Customer → not_penny⟨coin_entered⟩,
coin_entered → coin_return | button pressed w/ > .50 |
button pressed w/ < .50⟨wait for coins⟩ | not_penny⟨coin entered⟩,
⟨wait for coins⟩ → not_penny⟨coin entered⟩ | coin_return}

Figure 6. Grammar for the customer view.

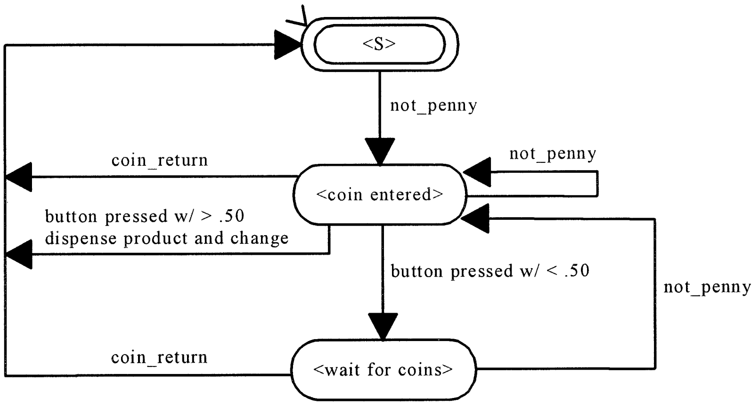

Figure 7. Conceptual state machine for the customer view.

Any errors found during this step cause the process to return to the first step and repeat until no errors are found. Verification has two steps. The first step is to check the grammar to ensure the scenario tree was constructed correctly. Verifying the grammar involves checking it for regularity, the absence of chain rules, and reachability. The second step is to verify the conceptual state machine for consistency and completeness. This is done by ensuring the conceptual state machine satisfies the following properties:

(a) it must be a deterministic finite-state machine (no one event can take the system to two different states) and

(b) there must be exactly one initial state and one terminating state for each scenario.

Scenario generation is used to generate scenarios in the verified formal model using the conceptual state machine. This is done by finding all the basis paths and the corresponding scenarios. A set of basis paths in the conceptual state machine is a set of paths in which each path traverses at least one new node or edge. The steps that generated scenarios for Fig. 4 are given in Fig. 8. Due to the loops present in Fig. 7, there is more than one set of valid basis paths.

1. not_penny, coin_return
2. not_penny, button pressed w/ > .50/dispense product and change
3. not_penny, button pressed w/ < .50, coin_return
4. not_penny, button pressed w/ < .50, not_penny, coin_return
5. not_penny, not_penny, coin_return

Figure 8. Generated scenario schemas for Fig. 7.

Prototype generation is the process of developing a scenario-based prototype derived from the conceptual machine constructed in the previous stage. Minimum work is needed to mimic the basic user interfaces and bind them to the scenario scripts. This becomes a simple and lively model, mimicking the behaviors of the expected software product.

Scenario validation is the process of using the system prototype to evaluate scenarios and demonstrate their validity to the user. If invalid scenarios are found, the process returns to the first step and repeats until all the scenarios are validated.

The scenario analysis process is finished when both the analyst and the domain expert are satisfied that the retained scenarios are a valid representation of the system that is to be created.

4. Scenario-based acceptance testing

A formal scenario-based acceptance test model for testing the external behavior of software systems during acceptance testing has been proposed by Hsia *et al.* [1994c]. This model comprises three submodels:

(1) a user view,

(2) an external system interface, and

(3) and external system view.

The user view model depicts the way a particular user group will use the system. This includes all the users' events and inputs as well as all the system responses and outputs. The user view represents the operational scenario scheme from the users' viewpoint. The external system interface view models the external behavior of the system's interactions with its external environment (hardware interfaces) in terms of inputs/outputs, events/actions, and responses. The external system view models external behavior by combining the user views and external interface views.

The following steps are taken to construct the scenario-based acceptance testing model. Please note that these are the first three steps of the scenario analysis process described in section 3.

(a) Scenario elicitation and specification. Different user views and different external interface views are elicited.

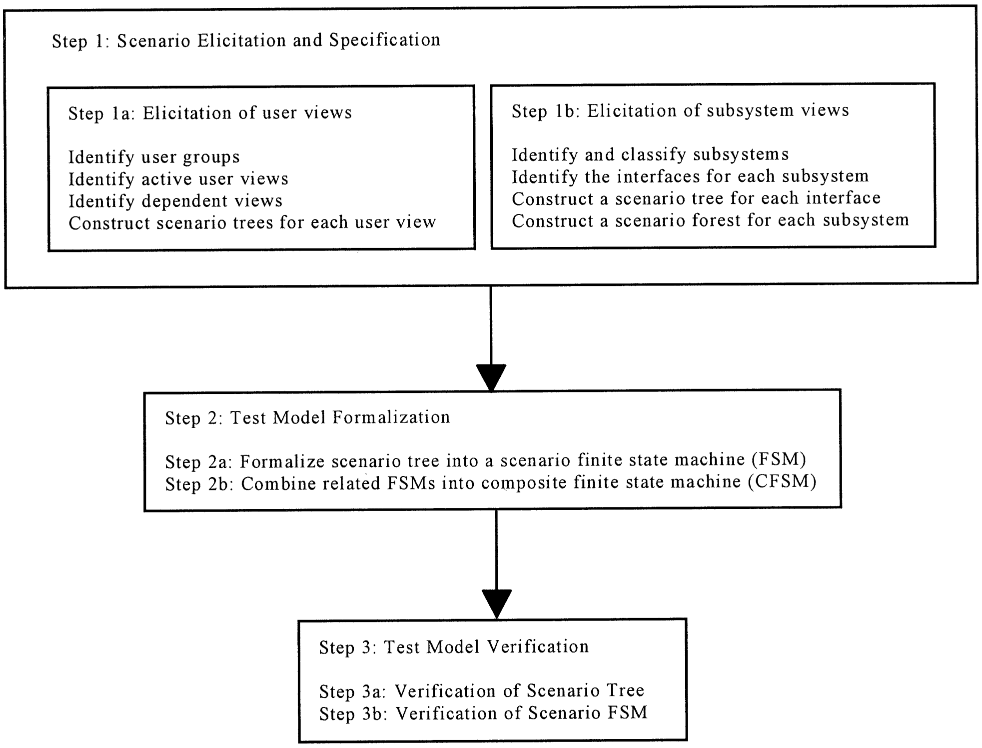

Figure 9. Scenario-based test model construction steps.

(b) Test model formalization. Each external interface view is formalized as a scenario-based finite state machine (FSM). These views are then merged together to form a system view. The system view is called a composite scenario-based finite state machine (CFSM).

(c) Test model verification. All FSM's and CFSM's are examined for determinism, consistency, correctness, and completeness.

Figure 9 is a pictorial representation of the scenario-based test model construction steps. In the following sections, each of these steps is discussed in more detail with reference to an example.

4.1. *Test model elicitation*

The elicitation of a test model for a software system consists of the following three steps:

(a) Elicit user views. This involves identifying and classifying all the various user views, events, event sequence invariants, and agents according to the types of user groups and building their respective scenario trees.

(b) Elicit external system interface views. This involves identifying external system interfaces according to the system requirements and constructing a scenario tree for each external system interface according to the comprehensive event trace in the system specifications.

(c) Combine the generated scenario trees into a scenario forest.

4.1.1. Constructing a scenario tree for a user view

Scenario trees are constructed by performing scenario elicitation for the various user views. A scenario tree consists of nodes and labeled edges. A node identifies a state of the system as seen by the user and does not necessarily have to be a system state. An edge represents the occurrence of an event type that transitions the system from one state to another.

4.1.2. An example

The following example shows how a formal scenario approach is applied to acceptance testing and how it improves the existing practice. The system used in this example is a model of the Data Transfer Equipment (DTE) used to load mission data onto the F-16. This model was developed to run on a test station. This example is kept simple while illustrating the concepts behind this approach. The requirements for this model are: The DTE model is responsible for sending mission planning data out on a multiplex channel (MUX). The model sits idle until it receives a command from the mux controller. A command consists of two parts, the file status (open, close) and the number of the file to process (1, 2, 3). If the DTE model receives a file type other than 1, 2, or 3, it outputs an invalid file type message and returns to an idle state, awaiting a new command.

In the DTE example, elicitation and scenario tree construction for the user view was performed in the following manner. After consulting requirements analysts and users, one user group was found; namely, the system testers. The active user view is the view of the system testers, and there are no dependent views. The scenario tree for the system test group's view was constructed by the tester as follows: the tester first constructed the root of the tree. He then consulted the users and discovered that the first event performed was specifying a request for file 1, 2 or 3 and then commanding the simulation to step one frame. For this, three edges were drawn from the root to the node (state). This procedure was continued until all the scenarios were elicited. Figure 10 shows the complete scenario tree for the system testers view. Other example constructions of scenario trees for a the user view can be found in [Hsia *et al*. 1994a, b].

4.1.2.1. Constructing a scenario tree for a system external interface. For any subsystem, its external interfaces are identified based on the system requirements. The construction of the scenario trees for the system external interface is based on

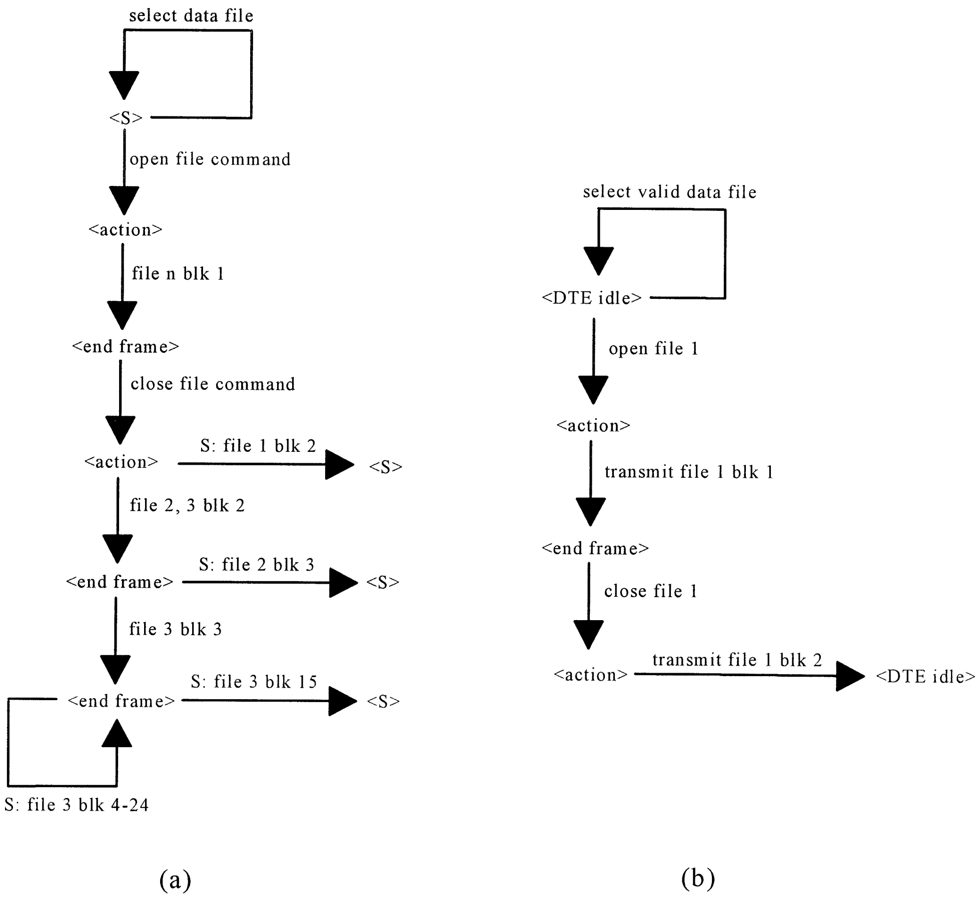

Figure 10. Scenario tree (a) and an instance (b) for user view.

a scenario trace mechanism. The scenario trace mechanism is an extension of Rumbaugh's event trace [Rumbaugh 1991] which has been modified to include the concept of interface to partition the interactions between various subsystems and users. For each interface defined, scenario elicitation is conducted in two steps: 1) the derivation of a scenario trace and 2) the construction of a scenario tree.

4.1.2.2. Derivation of scenario trace. Deriving the scenario trace involves modeling the interactions between any two subsystems in terms of event flows. Events in a scenario trace can be of any of the following types: input signal, output signal, or reactive internal action (any internal event that occurs as a response to another event).

A scenario trace model has at least one starting and one ending event and uses an OR operator to model the case in which only one of a set of events can occur.

The DTE example is used to illustrate the scenario trace model. Figure 11 shows the scenario trace model for the DTE system. The *simulation/MUX controller* interface interacts with the *DTE model* and the rest of the simulation. To construct the

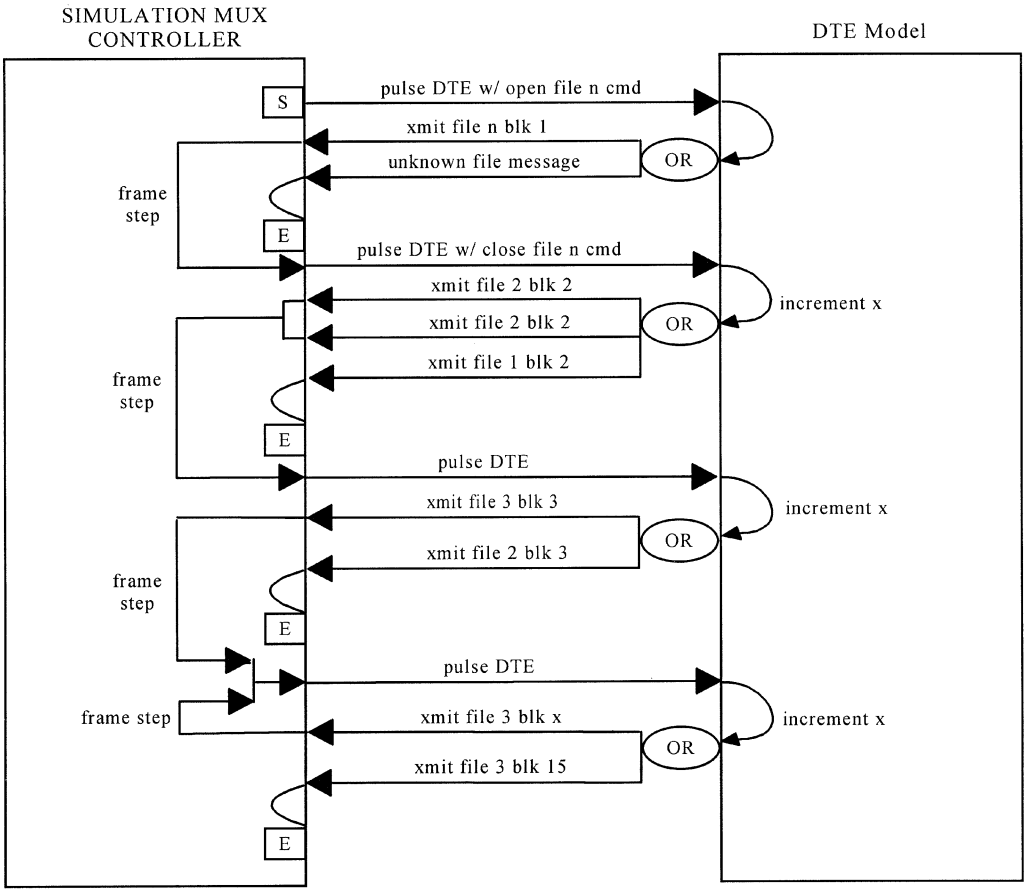

Figure 11. Scenario trace model for the DTE control.

event trace between the *DTE model* and the *simulation/MUX controller* interface, the requirement analyst identifies all the interacting event types (in this interface) from the requirements specifications (e.g., high level data flow diagrams of the interface). He then classifies the identified events as starting events, ending events, and intermediate events. In this example there is one starting event – *pulse DTE* with open file command and four possible ending events – *unknown file id*, *xmit file 1 blk 2*, *xmit file 2 blk 3*, or *xmit file 3 blk 25*. The rest of the events are intermediate events. Beginning with each starting event, the requirements analyst constructs the flow of events (including internal reactive actions) until an ending event is reached. Mutually exclusive events are depicted using an OR notation. In the example the starting event is a *pulse DTE* with open file command sent by the *simulation/MUX controller* through shared memory. The *DTE model* system will respond by sending the first block of the requested file or with an error message if an invalid file was requested. If the invalid file message was sent by the DTE, no more processing can be done so the scenario session ends. Similarly, the other possible responses are dealt with until an ending event is reached. This is similar to a depth first construction technique. The event sequences are continued until all the ending events are reached.

4.1.2.3. Scenario tree construction. The scenario tree for a system external interface and the scenario tree for the user view are very similar. Every node in the tree indicates the state of the subsystem interface at that point in the event trace and every edge is an event type. An edge is labeled in the format [Hsia *et al.* 1994b]:

Event := Event_item/Event_item | Event_item

Event_item := Conditional_Event_item | Unconditional_Event_item

Conditional_Event_item := [C]Unconditional_Event_item

Unconditional_Event_item := S : System event | S : A.System event
| user event | external system event

The notation for the event type *S*:*A.System* denotes that the event is performed by the system on the interface A. The absence of the A (i.e., event type *S*:*System*) denotes that the event is performed by the system on the current interface itself. The "/" is used to indicate the relationship between the triggering and triggered events. The "[C]" is used to represent the guard condition for an event. The scenario tree is constructed from the scenario trace model as discussed in section 4.1.2.1. The detailed construction outlined below is taken from [Hsia *et al.* 1994b]:

(a) First the root of the scenario tree is constructed.

(b) Each starting event in the scenario trace model becomes an edge from the root node. These edges form the first level branches of the tree. Every node in this first level indicates the state of the subsystem interface after the first event.

(c) Every subsequent event in the scenario trace model forms child edges in the tree. If there is an OR condition, the mutually exclusive events map into multiple edges from that node.

(d) Every edge corresponding to an ending event terminates in a leaf node of the tree. For the system interface of the DTE example, the scenario tree corresponding the scenario trace in Fig. 11 is shown in Fig. 12.

4.2. *Test model formalization*

The purpose of this step is to formalize a method to analyze, verify, and generate test scenarios in a systematic manner before proceeding with the actual acceptance testing. Test model formalization is accomplished in two steps. First, a scenario tree for an external system interface or a scenario tree for a user view is converted into a finite state machine (FSM). This FSM is called a scenario FSM and is the conceptual usage model of the system. Secondly, all the related FSMs that have been created are

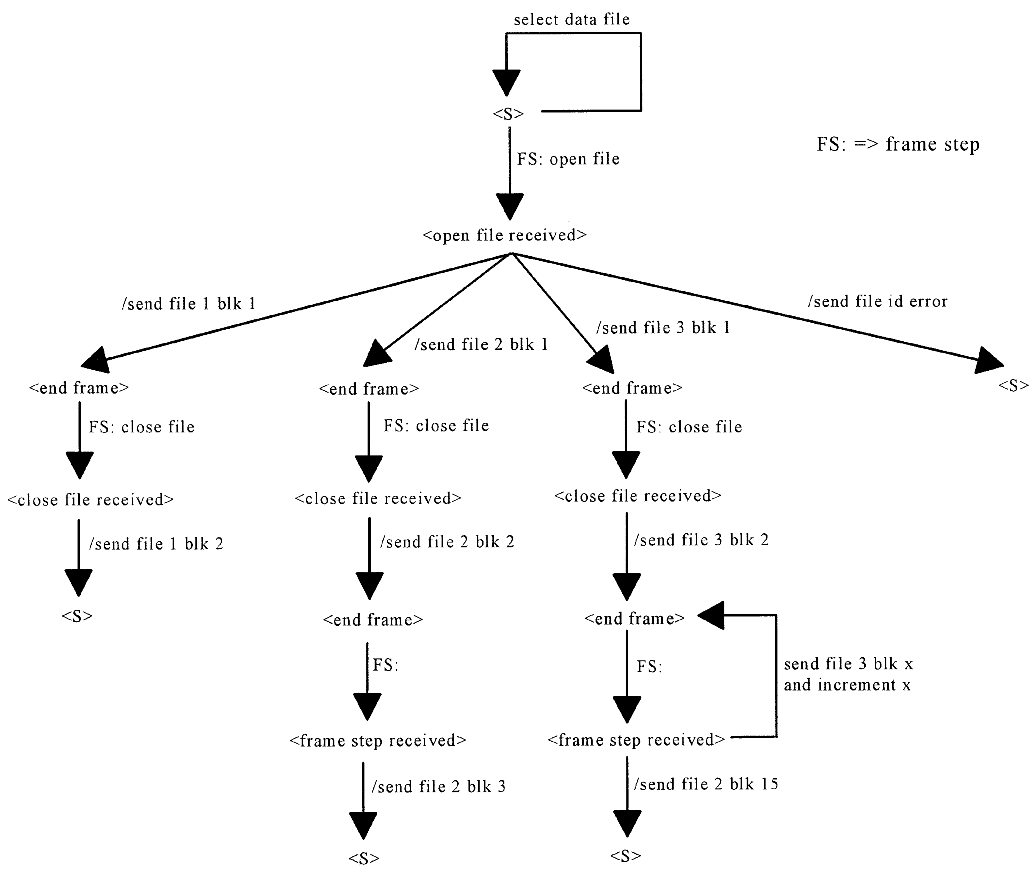

Figure 12. Scenario tree for DTE system interface.

combined to form one composite finite state machine (CFSM). This CSFM describes the whole subsystem.

The FSMs are used to model scenario schema and instances and to analyze, verify, and generate the test scenarios. The mapping rules used to formalize a scenario tree into a FSM are given in [Hsia *et al.* 1994b]. Given any FSM, a scenario scheme can be redefined as a basis path from the start state to the final state of the FSM. Thus, a sequence of specific events that covers a basis path of the FSM is a scenario instance. The FSM for the DTE system interface is shown in Fig. 13. Since there is only one interface in the DTE example, no CSFM is created.

4.3. Verification of a scenario-based test model

This step corresponds to the scenario verification portion of the scenario-analysis model described in section 3. A scenario-based test model is verified in two steps. First, all scenario trees are checked for completeness, inconsistency, and redundancy. This is done manually by the requirements analyst or tester and takes advantage of

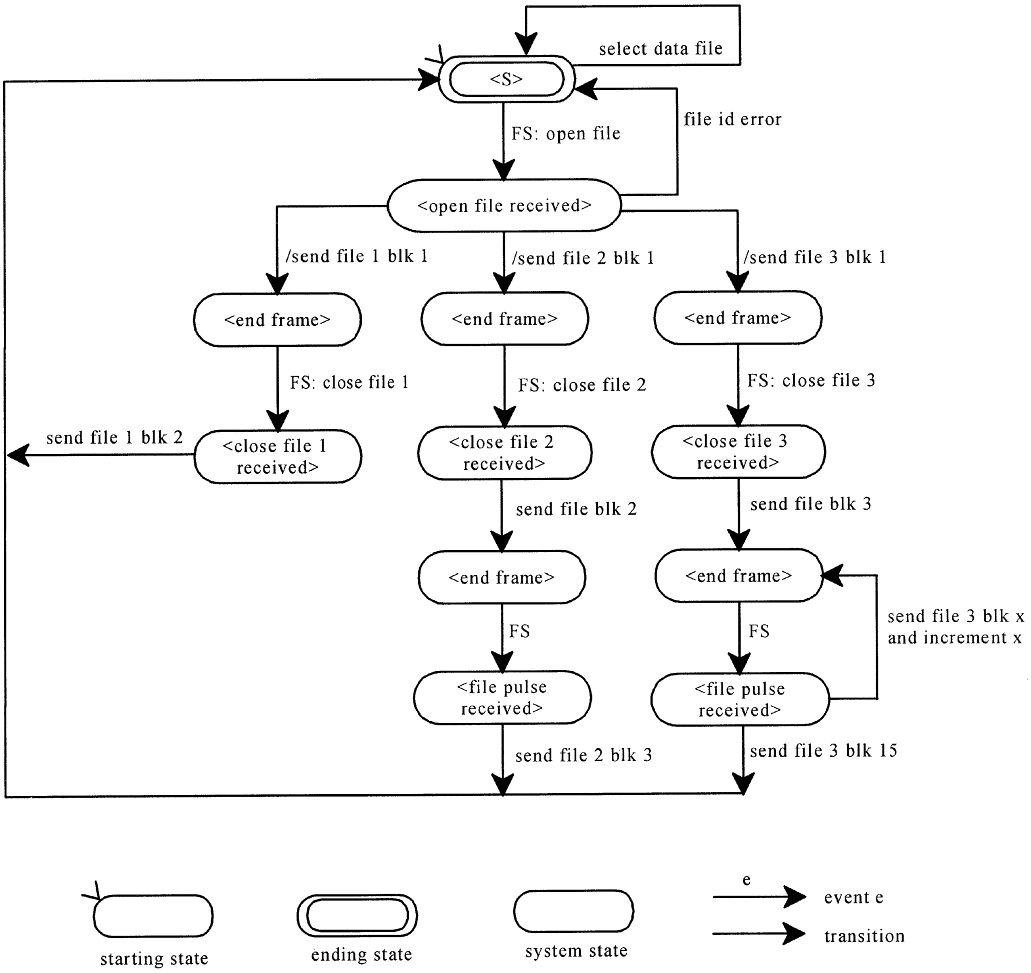

Figure 13. FSM for DTE system interface.

the domain knowledge of the users. Secondly, the scenario FSMs that were created are checked to ensure correctness and determinism. Correctness means the sequence of events is complete. The sequence is incomplete if the state of the system in not in the final state after the last event has occurred. A deterministic scenario FSM will not allow the same event to transition one state into two or more different states, it also demonstrates that there are no redundant and inconsistent scenarios.

4.4. Testing external behavior

The major focus of acceptance testing is on man-machine interactions, external interfaces, required functionality, and specified system constraints. To meet these objectives, acceptance testing of a software system consists of the following tasks:

- *Testing man-machine interfaces*. This is done to test the usability and reliability of the system and is accomplished by generating all the scenarios of an FSM, running them one at a time while checking the system's behavior. Errors found during this phase of testing are classified as user interaction errors. These include incorrect processing of user input and incorrect system responses.
- *Testing external system interfaces*. This is done to verify all the interactions between the system and its external system interfaces. This is accomplished by running through all the scenarios in the corresponding FSM. Errors uncovered here are external system interface errors which include incorrect processing of incoming messages or stimuli, incorrect outgoing messages or stimuli, and incorrect processing of different sequences of external events.
- *Testing the required functional features*. This demonstrates that each function specified in the requirements is implemented. Errors uncovered here are functional errors which include missing, extra, or incorrect functional features.
- *Testing the specified system constraints.* This test assures that the system meets the required safety and exception constraints. Errors detected here are classified as constraint and exception errors.

Each of these acceptance testing tasks satisfies certain criteria which are defined in [Hsia *et al*. 1994c]. These criteria are summarized below.

- *User view criterion.* This criterion ensures that for a given user view, there is at least one scenario instance that covers the basis path for that view's FSM.
- *External system interface criterion.* This criterion ensures that for a system interface FSM there is at least one scenario instance covering each basis path of the FSM.
- *Function criterion.* This criterion ensures that for each functional feature that is part of a set of functional features represented by a CFSM, there is a test set that covers all the basis paths associated with the functional feature (the associated FSM in the CSFM).
- *Specified constraint criterion.* This criterion ensures that for each system constraint which is part of a set of system constraints represented by a CFSM, there is a test set that covers all the basis paths that are associated with constraint (the associated FSM's in the CSFM).

4.5. Test generation

In order to cover all the scenario test criteria listed above, a scenario basis path generation method has been developed [Hsia *et al*. 1994b] that can be used to test the user interface and the subsystem interface. This step corresponds to the scenario generation portion of the scenario-analysis model described in section 3. The basic approach is to construct a basis path matrix Mbpath(k) for a set of related FSM's. The basis path matrix is a square matrix where each dimension is the number of states in the FSM. Each node and column corresponds to a node in the FSM. The elements of the matrix are basis path sets with each basis path having length k. After the matrix is constructed, it is used to generate the related scenarios. The formal definition of Mbpath(k) and its related operations as well as the method used to generate the scenarios are given in [Hsia *et al*. 1994b].

5. Benefits of using scenario analysis in acceptance testing

The major benefit of using a scenario approach for acceptance testing is its ability to allow users to work with requirements analysts in systematically specifying and generating acceptance test cases. Scenario analysis produces a set of sufficiently complete and consistent scenarios which can be used as the input for acceptance testing. Currently, developers must reintroduce requirements to derive test cases. This is time-consuming, subjective, and adds additional cost to a project. All of these shortcomings can be reduced or eliminated if scenarios are used as the basis for acceptance test case generation. With the present methods of generating acceptance test cases the developer or tester must be a domain expert to develop useful and meaningful test cases. With the scenario analysis method this necessity is removed thanks to user and domain expert involvement. Developers will benefit from this method by having access to acceptance test cases when they informally test their own code. Every developer has his or her own way to test code before it is released to formal testing. By knowing in advance the nature of the acceptance tests that will be run, the individual developer can create and conduct more meaningful tests, thus reducing the number of errors uncovered in formal testing.

Another advantage of this approach is its ease of use as the amount of training needed to use it would be minimal. The main technique used in this approach takes advantage of something humans already do naturally, which is to describe things in terms of scenarios. They only need to formalize their scenarios on paper. Some have argued that scenario analysis is just another form of a finite state machine. Although similar, scenario analysis has two major differences. One is that scenario analysis secures the users involvement from the beginning which helps develop clearer requirements and also leads to better acceptance test cases. It also provides an inconspicuous way to solicit users' participation throughout the project because scenarios are a natural means of communication between customers and developers. The other difference is that the finite state machine developed from scenario analysis become the users' conceptual usage model of the product. It is used in this case for black box testing, not for white box testing like other FSM's are used. In addition, it is possible to discuss usage completeness and test coverage because it has a usage model to compare with.

This approach has been applied to a safety-critical system, Therac-25 (a radioactive cancer treatment machine), and a comprehensive set of usage patterns were generated. We can create all the usage sequences from the model. The specific usage pattern which caused several fatal overdoses was identified in the test suite [Hsia *et* *al*. 1995]. It has a sequence length 20. Had this method been used in testing the Therac-25, many lives could have been spared. Apparently, this specific test case was not included in the test suite when the Therac-25 was tested for 2700 hours before it was released for commercialization. We have identified 2720 distinct test cases with sequence lengths less than or equal to 20 for this machine. We believe that this is not a single case in software industry and it also shows that the practice of acceptance testing is very much ad hoc and without any measurable concept of function/feature coverage. This new scenario-based testing is a systematic approach easily emulated by any testing group.

In [Szaboky 1996], a systematic approach for testing GUI applications is also developed following precisely the same concept. It points out that testing the functionality of a GUI based software is not any different from other older UI based one. Therefore, the scenario-based testing approach is likewise applicable. This method is suitable only for systems that have a single response to a stimulus, both of which are events. A user interface or database management system are good examples. This method is not suitable for complex systems such as a controller for multiple elevators or a flight control system because it cannot handle concurrent stimuli and responses. This limitation does not detract from its usefulness since in man-machine interaction, there is generally only one event response to a stimulus.

6. Conclusion

Through a review of literature on acceptance testing, a taxonomy of the current acceptance test approaches has been assembled and it has been established that acceptance testing is currently done in an ad hoc manner. A new approach based on scenario analysis has been presented which establishes a systematic approach to acceptance testing and can avoid many of the problems plaguing the current practice. The key features of this approach are that it is easy to use, amenable to automation, and effective at generating sufficiently comprehensive usage patterns. Starting from given the system requirements and domain experts knowledge, this new approach allows one to elicit, formalize, and verify an acceptance test model. This methodology also defines test criteria based on the test model and specifies scenario test generation methods that will generate the scenarios required to test each of the criteria. Users benefit from this approach because they can systematically generate test plans to achieve any desired test coverage while developers benefit by being empowered to verify test cases, automate tests, and increase test performance.

Acknowledgements

The material presented in this paper is based on work supported by the Texas Advanced Technology Program (Grant No. 003656-097), Fujitsu Network Transmission Systems, Inc., and the Software Engineering Center for Telecommunications at UTA.

References

Alford, M. (1977), "A Requirements Engineering Methodology for Real-Time Systems," *IEEE Transactions on Software Engineering 3*, 1, 60–69.

Andriole, S.J. (1989), *Storyboard Prototyping – A New Approach to User Requirements Analysis*, QED Information Systems, Wellesley, MA.

Bauer, J., S. Faasse, A. Finger, and W. Goodhue (1978), "The Automatic Generation and Execution of Function Test Plans for Electronic Switching Systems," In *Proceedings of the Software Quality Assurance Workshop*, ACM, New York, NY, pp. 92–100.

Bauer, J. and A. Finger (1979), "Test Plan Generation Using Formal Grammars," In *Proceedings of the Fourth International Conference on Software Engineering*, IEEE Computer Society Press, Los Alamitos, CA, pp. 425–432.

Beizer, B. (1984), *Software System Testing and Quality Assurance*, Van Nostrand Reinhold, New York, NY.

Chandrasekharan, M., B. Dasarathy, and Z. Kishimoto (1989), "Requirements-Based Testing of Real-Time Systems: Modeling for Testability," *IEEE Computer 18*, 4, 71–80.

Chow, T. (1977), "Testing Software Design Modeled by Finite State Machines," In *Proceedings COMPSAC'77*, IEEE Computer Society Press, Washington, DC, pp. 58–64; also in *IEEE Transactions on Software Engineering 4*, 3, 178–187.

Compton, H., P. Giloth, M. Haverty, and B. Niedfeldt (1977), "System Integration and Early Office Experience," *The Bell System Technical Journal 56*, 7, 1279–1294.

Dasarathy, B. and M. Chandrasekharan (1982), "Test Generation for Functional Validation of Real-Time Systems," Sixth IEEE International Conference on Software Engineering, Unpublished poster session.

Davis, A. (1980), "Automating the Requirements Phase: Benefits to Later Phases of the Software Life Cycle," In *Proceedings COMPSAC'80*, IEEE Computer Society Press, Washington, DC, pp. 42–48.

Davis, A. (1990), "System Testing: Implications of Requirements Specifications," *Information and Software Technology 32*, 6, 407–414.

Davis, A. (1993), *Software Requirements, Objects, Function & States*, Prentice Hall, Englewood Cliffs, NJ.

Davis, A. and T. Ranscher (1978), "Formal Techniques and Automatic Processing to Ensure Correctness in Requirements Specifications," In *Proceedings of the Specifications of Reliable Software Conference*, IEEE Computer Society Press, Los Alamitos, CA, pp. 15–35.

Deutsch, M. (1982), *Software Verification and Validation*, Prentice Hall, Englewood Cliffs, NJ.

Dunn, R. and R. Ullman (1982), *Quality Assurance for Computer Software*, McGraw-Hill, New York, NY.

Ferguson, R. and B. Korel (1996), "The Chaining Approach for Software Test Data Generation," *ACM Transactions on Software Engineering and Methodology 5*, 1, 63–86.

Gomaa, H. and D. Scott (1981), "Prototype as a Tool in the Specification of User Requirements," In *Proceedings of the 5th International Conference Software Engineering*, IEEE Computer Society Press, Los Alamitos, CA, pp. 333–342.

Hetzel, W. (1984), *The Complete Guide to Software Testing*, QED Information Systems, Wellesley, MA.

Hsia, P., J. Gao, J. Samuel, D. Kung, Y. Toyoshima, and C. Chen (1994a), "A Formal Approach to Scenario Analysis," *IEEE Software 11*, 2, 33–41.

Hsia, P., J. Gao, J. Samuel, D. Kung, Y. Toyoshima, and C. Chen (1994b), "Behavior-Based Integration Testing of Software Systems: A Formal Scenario Approach," In *Proceedings of the Third International Conference on System Integration*, IEEE Computer Society Press, Los Alamitos, CA, pp. 1138–1147.

Hsia, P., J. Gao, J. Samuel, D. Kung, Y. Toyoshima, and C. Chen (1994c), "Behavior-Based Acceptance Testing of Software Systems: A Formal Scenario Approach," In *Proceedings of COMPSAC'94*, IEEE Computer Society Press, Los Alamitos, CA, pp. 293–298.

Hsia, P., J. Samuel, D. Kung, L. Li, C.T. Hsu, C. Chen, and Y. Toyoshima (1995), "A Usage-Model Based Approach to Test the Therac-25," *Safety and Reliability in Emerging Control Technologies*, IFAC Workshop Report, Pergamon Press, Daytona Beach, FL, pp. 55–63.

Huebner, D. (1979), "Systems Validation through Automated Requirements Verification," In *Proceedings of COMPSAC'79*, IEEE Computer Society Press, Washington, DC, pp. 222–227.

Janowiak, T. (1990), "A Simulation for Combat Systems Development and Acceptance Testing," In *Proceedings of the 1990 Winter Simulation Conference*, IEEE, New York, NY, pp. 210–213.

Kaner, C., J. Falk, and H. Nguyen (1988), *Testing Computer Software*, Second Edition, Van Nostrand Reinhold, New York, NY.

Kleinstein, D. (1988), "System Acceptance Testing in the Real World: A Case Study," In *Proceedings of the Third Israel Conference on Computer Systems and Software Engineering*, Tel-Aviv, Israel, pp. 40–46.

Krause, K. and L. Diamant (1978), "A Management Methodology for Testing Software Requirements," In *Proceedings of COMPSAC'78*, IEEE Computer Society Press, Washington, DC, pp. 749–754.

Lea, R., S. Chen, and C. Chung (1991), "On Generating Test Data From Prototypes," In *Proceedings of the 15th International Computer Software and Applications Conference*, IEEE Computer Society Press, Los Alamitos, CA, pp. 345–350.

Linehan, T., Jr. (1988), "Application Software Configuration Management and Testing in a Pharmaceutical Laboratory Automation Environment," In *Proceedings of the Conference on Software Maintenance*, IEEE Computer Society Press, Los Alamitos, CA, pp. 178–182.

Lipow, M. and T. A. Thayer (1977), "Prediction of Software Failures," In *Proceedings of the Annual Symposium on Reliability and Maintainability*, IEEE, New York, NY, pp. 489–494.

McCabe, T. (1983), "Structured Requirements and Testing," *Software Outlook Newsletter*, McCabe and Associates, Columbia, MD.

McCabe, T. and G. Schulmeyer (1985), "System Testing aided by Structured Analysis – A Practical Experience," *IEEE Transactions on Software Engineering 11*, 9, 917–921.

Meyers, G. (1979), *The Art of Software Testing*, Wiley, New York, NY.

Musa, J. and A. Ackerman (1989), "Quantifying Software Validation: When to Stop Testing," *IEEE Software*, 19–27.

Nardi, B. (1992), "The Use of Scenarios in Design," *SIGCHI Bulletin 24*, 4, 13–14.

NBS (1984), *Guideline for Lifecycle Validation, Verification, and Testing of Computer Software* (1984), US Department of Commerce, National Bureau of Standards, Springfield, VA.

Osder, S. and D. Decker (1982), "Software Validation for Flight-Critical Systems," In *Proceedings of the Phoenix Conference on Computers and Communications*, IEEE Computer Society Press, Washington, DC, pp. 193–198.

Protzel, P. (1988), "Automatically Generated Acceptance Test: A Software Reliability Experiment," In *Proceedings of the 2nd Workshop on Software Testing, Verification, and Analysis*, IEEE Computer Society Press, Washington, DC, pp. 196–203.

Quirk, W., Ed. (1985), *Verification and Validation of Real-Time Software*, Springer, Berlin, Germany.

Ramamoorthy, C., S. Ho, and W. Chen (1976), "On the Automated Generation of Program Test Data," *IEEE Transactions on Software Engineering 2*, 4, 293–300.

Ray, S. (1989), "Acceptance Tests of Distributed Processing Oriented Supervisory Control and Data Acquisition (SCADA) Systems for Offshore Platforms," In *Proceedings of the First European Test Conference*, IEEE Computer Society Press, Washington, DC, pp. 221–228.

Rumbaugh, J. (1991), *Object-Oriented Modeling and Design*, Prentice-Hall, Englewood Cliffs, NJ.

Samson, D. (1990), "Test Planning from System Requirements: A Classification Approach," *Jet Propulsion Laboratory Quarterly Report 12*, 33–37.

Szaboky, G. (1996), "A Systematic Approach for Testing GUI Applications," Masters Thesis, Computer Science and Engineering Department, University of Texas at Arlington, Arlington, TX.

Tausworthe, R. (1979), *Standardized Development of Computer Software, Part II, Standards,* Prentice-Hall, Englewood Cliffs, NJ.

Wallace, D. and J. Cherniavsky (1986), Report on the NBS Software Acceptance Test Workshop April 1–2, 1986, Natl. Bur. Stand. US Department of Commerce, Washington, DC.

Wallace, D. (1990), Guide to Software Acceptance, US Department of Commerce, National Institute of Standards and Technology, Gaithersburg, MD.

Wardle, P. (1991), "Methodology and Tools for Requirements Capture, Traceability, and Verification," In *Proceedings of the Third International Conference on Software Engineering for Real Time Systems* (*Conf. Publication No. 344*), The Institute, London, UK, pp. 46–50.

Worrest, R. (1982), "Using an Automatic Test Executor to Support Parallel System and Test Development," In *Proceedings of the Phoenix Conference on Computers and Communications*, IEEE Computer Society Press, Washington, DC, pp. 204–208.