

# IMPACT OF SEQUENCE-BASED SOFTWARE SPECIFICATION ON STATISTICAL SOFTWARE TESTING

S. J. PROWELL

**ABSTRACT.** The combination of sequence-based software specification with statistical software testing yields direct benefits of reduced start-up and communication overhead and the potential for automated generation of initial usage models and test oracles. Indirect benefits include better developer and system engineer understanding of external usability issues and an emphasis on external events which supports evaluating testability of requirements. This paper introduces the sequence-based specification techniques of sequence enumeration and sequence abstraction, then proceeds to trace the impacts of sequence-based specification on the development of test plans, usage models, and testing oracles.

## 1. MOTIVATION

Statistical testing of software based on a usage model requires that test engineers precisely define the test boundary, extract the expected usage profile, legal inputs, potential outputs, valid sequencing of inputs, and expected software responses to given input sequences. Extracting these details requires considerable communication between testing team members and both software developers and systems engineers. Misunderstandings about fundamental details of the software's input space and intended function often result in significant revision of software test plans and usage models. Further, software developers and systems engineers may communicate specialized knowledge of the software's implementation which may

---

*Key words and phrases.* Specification, sequence-based specification, statistical testing, usage models.

Dr. Prowell is with Q-Labs, Inc., 5516 Lonas Road, Suite 110, Knoxville, TN 37909, USA. Email: Stacy.Prowell@Q-Labs.com.

bias software testing. Often such knowledge rests on the assumption that the software system is correctly implemented (for example, the claim that some behavior is “impossible” because the software is intended to make it so), and may therefore hinder the utility of the software testing phase.

The introduction of sequence-based software specification can directly reduce the start-up costs incurred during testing by requiring that the software’s input domain, valid input sequences, and intended response for each sequence be considered early. Though additional costs are incurred during the specification phase, all subsequent phases of software development and testing benefit from the early introduction of a detailed, external specification. This reduces start-up times and communication overhead for all phases by helping identify and confront potential risks early in the process, and pays dividends in software testing.

## 2. SEQUENCE-BASED SPECIFICATION

Sequence-based specification is a collection of techniques for rigorous development of an external, functional specification of a software system’s intended behavior, known as a *black box specification* [2]. The development of this black box specification is guided by the technique of *sequence enumeration*, which is the literal enumeration of sequences of inputs and the assignment of correct responses to each enumerated sequence. Sequence enumeration is controlled and focused through the development of *sequence abstractions*, which allow practitioners to change their view of the system’s inputs and outputs without losing the formal nature of the resulting specification.

Sequence-based specification proceeds as follows:

1. The system boundary is specified by defining all software interfaces as precisely as possible.
2. All direct inputs to the system (stimuli) are identified.
3. Sequences of stimuli are enumerated, and a response is given for each sequence. All choices are justified by tracing to either

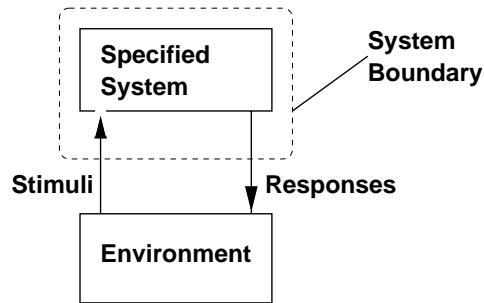


FIGURE 1. System boundary and environment

the software requirements, or by the introduction of a derived requirement.

4. Abstract stimuli are defined and introduced into the enumeration process as necessary to control growth, hide details, or refocus the enumeration activity.
5. From the enumerations, a black box specification is constructed and refined to the appropriate level of detail.

This section contains a short review of sequence-based specification. More detail may be found in [2], with full details and a case study in [3]. A more extensive case study will be available in [4]. Proofs of the theorems presented in this section can be found in [1].

**2.1. Identification of the System Boundary.** The system boundary lists all system interfaces, and should be documented in an interface specification. The collection of external entities with which the system communicates directly makes up the system's *environment*. Events which occur in the environment, and which can affect system behavior are *stimuli*. Events which occur in the system and which are observable from the environment are *responses*. Note that this definition of stimulus and response includes changes to monitored and controlled variables. The system boundary is depicted in fig. 1.

Work on a black box specification cannot begin until an initial list of stimuli and responses is in hand, though this list may be revised as work proceeds. In fact, the process of sequence enumeration, based as it is on a deterministic view of the system, may reveal problems in

the system boundary definition, resulting in revisions of the interface specification.

**2.2. The Black Box Function.** Let the set of all system stimuli be denoted  $S$ , and let the set of all system responses be denoted  $R$ . The black box specification denotes a complete function  $BB : S^* \rightarrow R$ , where  $S^*$  denotes all finite-length sequences of stimuli, which are interpreted as stimulus histories from left to right. Each sequence of stimuli is mapped to a unique value in  $R$ , which is the intended response to the most recent stimulus in the sequence. Software in operation might emit a response for every stimulus in a sequence, but the black box specification identifies only the last response.

Throughout the rest of this paper, let  $\lambda$  denote the empty sequence, and let sequence concatenation be denoted by juxtaposition, so the concatenation of  $u \in S^*$  onto the left end of  $v \in S^*$  is denoted  $uv$ .

The mapping rule of the specification requires that all sequences be assigned a unique value from  $R$ , but some sequences may not be physically realizable for software in operation because they violate the definition of the system. Consider a sequence which has several stimuli prior to the software being invoked for the first time. Such a sequence cannot be observed by the software. Such sequences are called *illegal*, since they violate the system definition. To account for these sequences,  $R$  is extended to include a special value  $\omega$ , to which all illegal sequences are mapped. Software in operation need not generate an externally-visible response for every stimulus sequence, but the mapping rule requires a value from  $R$ . To accommodate these sequences,  $R$  is further extended to include a special “null response” denoted  $0$  and interpreted as “no externally-visible response.”

**2.3. The Sequence Enumeration Method.** Sequence enumeration is a technique for discovering the black box function for a system

through the literal enumeration of all sequences of stimuli. In sequence enumeration, practitioners generate sequences from  $S^*$  in order by length, and then according to some fixed ordering among sequences of a particular length. As each sequence is generated, practitioners assign some value from  $R$  for the sequence, tracing the choice to software requirements or documenting derived requirements as necessary to justify their decisions. This systematic process increases developer understanding of the system's external behavior.

During sequence enumeration, practitioners may discover that all extensions of one sequence are mapped to the same response as the identical extensions of the other sequence. Formally, two sequences  $u$  and  $v$  are (*Mealy*) *equivalent*, denoted  $u \equiv_{\rho_{Me}} v$  if and only if  $\forall w \in S^*, w \neq \lambda, \text{BB}(uw) = \text{BB}(vw)$ . If, in an enumeration, the current sequence  $u$  is equivalent to a previously-enumerated sequence  $v$ , then practitioners note both the value  $\text{BB}(u)$  and the equivalence to previous sequence  $v$ . In this case,  $u$  is *reducible* to  $v$ . If there are no previous sequences to which  $u$  may be reduced, then  $u$  is *irreducible*.

Henceforth, the mapping of sequence  $u$  to response  $r \in R$  in an enumeration will be denoted  $u \mapsto r$ . If  $u$  is found to be reducible to previous sequence  $v$ , then this is denoted  $u \mapsto r / \equiv v$ .

**Theorem.** [*Canonical Sequence*] Let  $u \in S^*$  be a sequence. Then there exists a unique sequence (the “normal form”)  $v \in [u]_{\rho_{Me}}$  which is irreducible.

The unique normal forms for the reduction system defined by the equivalence  $\rho_{Me}$  are called *canonical sequences*. Since equivalence relations are transitive, and since every equivalence class has a canonical sequence, every reduction in an enumeration is required to be to an unreduced sequence.

Enumerations are generated by extending sequences of length  $n$  to obtain all sequences of length  $n + 1$ . If a sequence is illegal ( $u \mapsto \omega$ ), then all sequences with prefix  $u$  are also illegal, and thus illegal sequences need not be extended in an enumeration. If a sequence is reduced to previous sequence  $v$ , then for any sequence  $uw$  ( $w \neq \lambda$ ),  $\text{BB}(uw) = \text{BB}(vw)$ . Thus sequences which have been reduced need

not be extended in an enumeration. It follows that only unreduced, legal sequences need to be extended. If no sequences of some length  $n$  need to be extended, then the enumeration is *complete*.

**Theorem.** *A complete enumeration specifies a response for every sequence  $u \in S^*$ .*

A sequence enumeration thus defines a complete, consistent black box specification for a software system.

Every enumeration can be viewed as a Mealy machine by taking the canonical sequences as states. This leads to a direct conversion of the enumeration, and thus the black box, into a state machine.

**2.4. Controlling Enumeration Through Sequence Abstractions.** Sequence enumeration is made practical through the application of abstraction techniques, which can be used to control growth, defer details without losing them, and change views of the system. A sequence abstraction is a complete mapping  $\phi : X^* \rightarrow Y^*$  from *atomic* (stimulus) sequences  $X^*$  to *abstract* (stimulus) sequences  $Y^*$  such that the mapping satisfies two properties:

- abstract sequences are never longer than corresponding atomic sequences:  $\forall u \in X^*, |\phi(u)| \leq |u|$ , and
- the order of the stimuli in the sequence is preserved:  $\forall u, v \in X^*, \phi(u)$  is a prefix of  $\phi(uv)$ .

There are several common forms of abstractions. Some of these “abstraction patterns” are considered in [1] and [4]. The process by which one defines abstractions is beyond the scope of this paper. Fig. 2 illustrates the idea of abstractions. Here a sequence of request connection and accept connection request events, denoted  $R(n)$  and  $A(n)$ , respectively, is transformed into a sequence of connect events, denoted  $C(n)$ .

Abstractions may be used as follows, in conjunction with enumeration:

1. Enumerate sequences at the lowest reasonable level of abstraction.

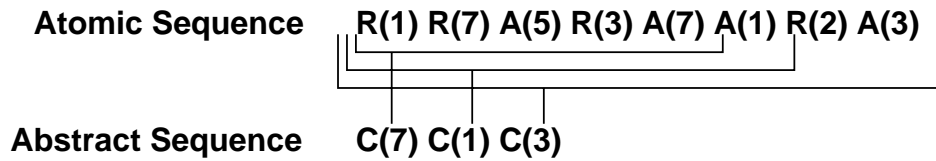


FIGURE 2. An abstract view of a sequence

2. If work stalls or is found to be unproductive, replace atomic stimuli with abstract stimuli to resolve the difficulty and restart enumeration with the abstract stimulus set.
3. Continue to invent abstract stimuli and enumerate until a complete enumeration is obtained, or until system behavior is sufficiently understood to write a complete, consistent black box specification at some level of abstraction.

### 3. STATISTICAL TESTING OF SOFTWARE AND THE IMPACT OF SEQUENCE BASED SPECIFICATION

Statistical testing refers to the application of statistical science to software testing, and may involve both random and non-random methods. The particular kind of statistical testing to be discussed here involves the generation of software test cases from a stochastic model representing the known or expected usage of a software system (a *usage model*). The stochastic model used will be a finite-state, discrete parameter, irreducible Markov chain. The application of such a model to the testing of software has been widely discussed (for example [5], [6], and [7]). For this reason, the presentation of statistical testing here will be terse.

As with the system specification, the intent is to have an external, black box view of the software to the extent possible. Knowledge of the internal implementation of a system is a source of bias in the statistical experiment, and can result in failure to test portions of the system, or in the direction of significant effort to seldom-used, non-critical functionality. Because the software specification is developed with an external, implementation-independent view, this focus on external events is easier to achieve. Additionally, it provides a means

for discussions between developers and test practitioners while reducing the risk of introducing an implementation bias.

### 3.1. Test Planning.

3.1.1. *Test Boundary.* Software testing can be viewed as a statistical experiment. One cannot run all possible tests, therefore only some sample of the potential tests will be run, and a conclusion about the reliability of the software in operation use must be derived from the software's performance on this sample. Treating testing as a statistical experiment implies that testing must be a well-defined procedure performed under specified operating conditions in order to generate useful, accurate conclusions.

The system under test must be understood and isolated by the definition of a *test boundary*. The test boundary consists of the precise definition of all interfaces with the system under test. This is similar to the system boundary, with the following exceptions: for every interface which is cut, all stimuli must be controllable, and all responses must be observable, throughout the test. For this reason, practitioners may choose a test boundary which includes some components which are in the system's environment. The test boundary is illustrated in fig. 3.

The definition of the test boundary, which includes the software's interfaces, is crucial to the rest of testing. Inputs to the system which are unknown to the test team may escape testing; vague or incorrect interface descriptions will result in long discussions with developers (who may argue for or against testing certain parts of the system, because of their prior assumptions about system reliability) and in wasted effort due to misinterpreting the interface specifications.

Sequence-based specification requires the development and maintenance of a complete system boundary definition. The process of enumeration and the deterministic nature of the specification force developers to insure that external information sources critical to the production of software responses are documented. This insures the

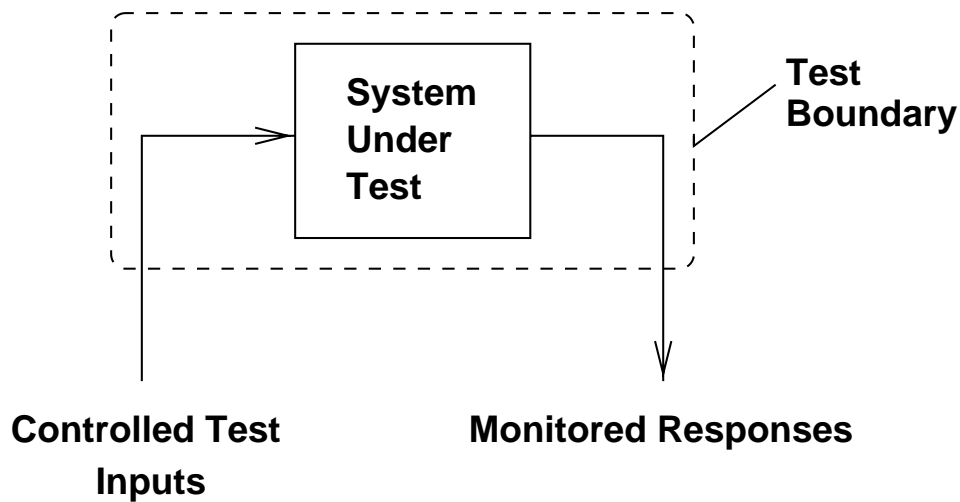


FIGURE 3. Test boundary

persistence of valid interface specifications which may be made available to test practitioners. Further, precise interface descriptions reveal what system stimuli and responses can be controlled and monitored effectively during a test. The availability of a precise interface description serves both to reduce communication overhead between test practitioners and developers, and focuses the discussions on the external system interfaces, thus reducing the chance of introducing an implementation bias into the testing phase.

The most significant advantage gained from such an interface specification is the ability to effectively plan for automation. Since interfaces are precisely specified, test drivers and monitors may be developed in parallel to the software development effort. In this way start-up time, and thus cost, for software testing is significantly reduced.

During test planning, one must also consider requirements coverage. Sequence-based specifications directly support requirements traceability through sequence enumeration. It is thus possible to identify all input sequences which implement a given requirement. This allows assessing whether the requirement is testable given the

external view and chosen test boundary. The ability to quickly transform a requirement into a collection of sequences also provides for the use of non-random, crafted test cases to address specific requirements.

3.1.2. *Test Stratification.* The testing effort may be divided and focused along three dimensions: types of users, types of uses, and environments of use. A combination of these three dimensions which is valid for software testing is called a *stratum*. Identification of test strata and the assignment of testing budget and effort to each identified stratum is *stratification planning*. The idea of test planning is illustrated in fig. 4.

For the purpose of stratification planning, a *user* is any source of stimuli to the system under test, and these may be deduced from the interface specification. A *use* is any single instance of software usage, and may be defined in terms of specific start and finish events, number of commands, duration of test, or simply by specifying starting and terminating conditions. A use always begins in some initial state (perhaps one of several), and ends in some defined final state (perhaps one of several). The initial and final states of any such use must be defined and *must be verifiable*. If a test is intended to leave the system under test in some particular state (such as terminated, with temporary files deleted) but instead leaves the system in some different state (unterminated, or with temporary files not deleted), then the test must be counted as a failure [8].

The identification of appropriate initial and final states is greatly aided by access to a consistent, complete specification. As a simple example, consider the definition of the initial state to be “uninvoked,” and the definition of the final state to be “terminated.” Under what conditions does the software terminate? This can be found immediately by examining the specification.

Examination of the specification will reveal important classes of use which might otherwise be missed in testing, or tested insufficiently. For example, it is common to discover previously-unconsidered

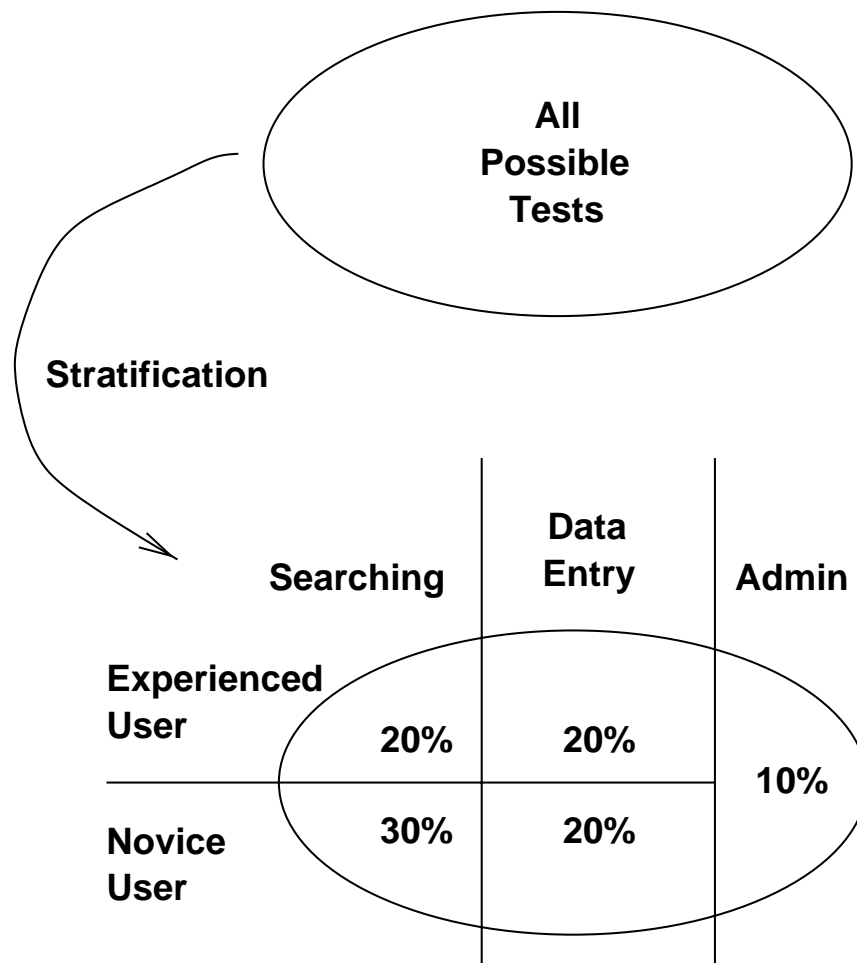


FIGURE 4. Stratification

critical behavior (such as error conditions) as one develops a sequence enumeration. The resulting specification is passed to testing, and additional testing effort can be directed to this behavior. This redirection of test effort occurs as the result of examining an external behavioral specification and not due to knowledge of internal implementation, and is thus based on user-perceived function and risk.

**3.2. Modeling Expected Software Usage.** Software use will be modeled as a finite-state, discrete parameter, irreducible Markov chain. Such a model is essentially a finite state machine whose transitions

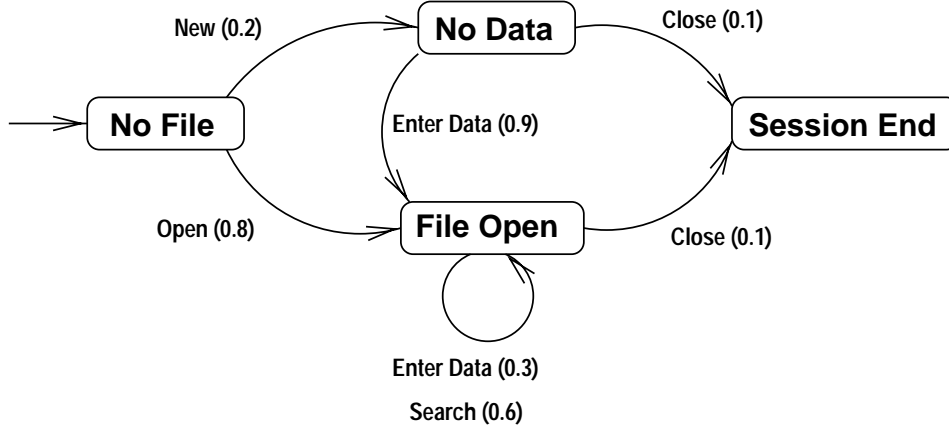


FIGURE 5. A Markov chain usage model

are labelled with stimuli and have associated probabilities. A graphical view of such a model is given in fig. 5, with initial state “No File.” States in this model correspond to *states of use*. Let any path in the chain which begins with the initial state be a *test trajectory*. For example, one test trajectory for the model in fig. 5 is “Open, Search, Search, Enter Data, Close” with probability 0.00864. Any test trajectory which terminates in a final state is therefore a *test case*; thus the trajectory just discussed is also a test case, since “Session End” is the final state. Let  $\rho_U$  be a relation on the set of test trajectories  $S^*$ , and let  $p(u)$  denote the expected probability in use of test trajectory  $u$ . Two test trajectories  $u$  and  $v$  are *usage equivalent*, denoted  $u \equiv_{\rho_U} v$  if extensions are always approximately equally likely. In short, the likelihood of future use is independent of which sequence has been observed.

Formally, choose some small  $0 < \epsilon < 1$  and break the interval  $[0, 1]$  into  $[0, \epsilon), [\epsilon, 2\epsilon), \dots, [n\epsilon, 1]$ . Then two values are  $\epsilon$ -equivalent, written  $p \approx_{\epsilon} q$  if and only if they fall in the same interval. Equivalence of test trajectories is given by:  $u \equiv_{\rho_U} v$  if and only if  $\forall w \in S^*, p(uw) \approx_{\epsilon} p(vw)$ . Note that this relation is an equivalence relation, and we may use it to deduce a state space from  $S^*$ . Formally, a *state of use* is an element of  $S^* / \approx_{\epsilon}$ .

This kind of trajectory equivalence is similar to the notion of sequence equivalence used in sequence-based specification. While there is no simple containment of one in the other, there is a weak relationship which can be exploited. A response  $r \in R$  may change the potential input domain of the software (perhaps by opening or closing a window, or turning on or off some external hardware), and clearly two sequences which have differing future input domains are unlikely to be usage equivalent. Thus if two sequences are not Mealy equivalent, then they are unlikely to be usage equivalent. It follows that different states of the Mealy machine derived from an enumeration are likely to correspond to different states of use. Thus test practitioners may take the Mealy machine as the first cut of a usage model.

The ability to automatically generate an initial usage model from the software specification is a tremendous advantage to test practitioners. The development of a usage model requires considerable knowledge of both the software's legal sequences and generated responses, both of which are directly encoded in the Mealy machine. Practitioners may then modify the machine by combining states to reduce test overhead and model size, or by splitting states based on usage criteria not considered in the specification (such as time of day, type of usage, whether a file has just been printed, etc.). Thus the test practitioners are able to quickly shift their focus to optimizing their efforts to achieve test goals.

**3.3. Execution of Tests and the Role of Oracles.** The role of a test oracle is illustrated in fig. 6. The Mealy machine additionally encodes the associated software response, and thus can serve as a test oracle (up to the level of abstraction used in the specification). As sequences of inputs are generated for input to the software, one can use the Mealy machine to predict the appropriate response. This response may then be checked against the response generated by the software.

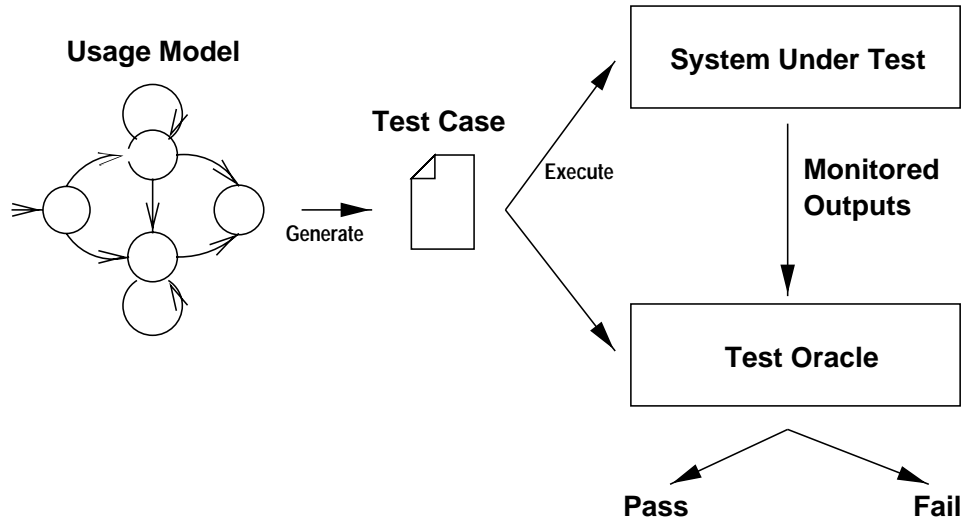


FIGURE 6. Test oracle

Two factors complicate the use of the Mealy machine as an oracle. First, abstractions may be used in the development of the enumeration, and thus in the Mealy machine. These can be removed through function composition, as described in [3]. One can implement the abstractions and, given the sequence of real inputs passed to the software, determine the appropriate abstract sequence. The Mealy machine will then reveal the abstract response, which can be mapped to the particular atomic response.

Second, and more significantly, the test boundary and the system boundary may differ. As a result, some software inputs needed to determine a particular response given the Mealy machine may be unavailable to the oracle. One solution is the following. Transitions in the Mealy machine which correspond to such unavailable stimuli are changed to  $\lambda$ -transitions (i.e., transitions which may be taken nondeterministically). Thus for any sequence executed against the software, the sequence prefix reveals that the software must be in one of a collection of states. The final stimulus is then used to determine all potential responses. Note that, because of the nondeterministic nature of this oracle, it is possible that failures will not be

detected, as pointed out in [9]. Note that, because some stimuli are lost, no other solution is possible.

#### 4. CONCLUSIONS

Aside from the obvious advantages to test practitioners of a any software specification the development of a sequence-based specification has the following direct benefits to software testing:

- The precise nature of the specification reduces communication overhead by conveying the software's intended behavior unambiguously
- Requirements traceability, maintained throughout the sequence enumeration process, simplifies the construction of crafted test cases to address specific requirements by identifying the sequences which implement a particular requirement
- The precise system boundary definition reduces start-up time and provides early opportunities to assess and develop test drivers and scaffolding
- Using the Mealy machine from the enumeration as an initial Markov model structure reduces Model construction time
- The Mealy machine from the enumeration serves as a test oracle to support automated testing, reducing the overall cost of automating software testing

These benefits all serve to either reduce the effort required during software testing, or to improve management of the testing process.

Software testing benefits indirectly from the development of a sequence-based specification in the following ways:

- The focus on external events reduces the chance of implementation bias during testing, providing improved experimental control
- The strict definition of software interfaces allows improved assessment of software testability and appropriate test boundary
- Understanding of the system is improved for developers, systems engineers, and test practitioners

The application of sequence-based specification on industrial projects has resulted in the development of new sequence-based practices and the extension of existing practices to meet the needs of specific software domains, including real-time software, distributed systems, and dynamic network environments. These additions to the theory will have implications with respect to software testing which should be explored further. For example, one common extension of the basic enumeration practice is to allow stimuli to interrupt the production of a response. This has implications with respect to testability which must be considered in a software test plan.

Abstractions are used in the development of Markov chain usage models to control the growth of the state space. The relationship of these abstractions to the abstractions defined during analysis of software behavior should be explored. At present, it seems that abstractions developed during specification are often equally useful during development of a usage model.

The use of the enumeration Mealy machine as a test oracle requires further development. In particular, the problem of how to deal with nondeterministic behavior during testing must be addressed. If the communication which is obscured is with a component which receives no stimuli from other external systems, and if a sequence-based specification is available for the component, then the component may be modeled by a second Mealy machine, and the nondeterminism eliminated. This would increase the accuracy of the test oracle, and thus yield a more effective software test.

## REFERENCES

- [1] S.J. Prowell, "Sequence-Based Software Specification," (Dissertation), University of Tennessee, Knoxville, Tennessee, 1996.
- [2] S.J. Prowell and J.H. Poore, "Sequence-Based Software Specification of Deterministic Systems," *Software—Practice and Experience*, v. 28, n. 3, March 1998, pp. 329-344.

- [3] S.J. Prowell, J.C. Martin, C.J. Trammell, J.H. Poore, *Cleanroom Software Engineering: A Practitioners Guide* (version 2.0), Knoxville, Tennessee: Q-Labs, Inc., 1998, Chapter 3.
- [4] S.J. Prowell, C.J. Trammell, R.C. Linger, J.H. Poore, *Cleanroom Software Engineering: Technology and Process*, Reading, Massachusetts: Addison-Wesley-Longman, to appear fall 1998.
- [5] J.A. Whittaker and M.G. Thomason, "A Markov Chain Model for Statistical Software Testing," *IEEE Transactions on Software Engineering*, v. 20, n. 10, October 1994, pp. 812-824.
- [6] G.H. Walton, J.H. Poore, and C.J. Trammell, "Statistical Testing of Software Based on a Usage Model," *Software—Practice and Experience*, v. 25, n. 1, January 1995, pp. 97-108.
- [7] W.J. Gutjahr, "Importance Sampling of Test Cases in Markovian Software Usage Models," *Probability in the Engineering and Information Sciences*, v. 11, 1997, pp. 19-36.
- [8] C.J. Trammell and J.H. Poore, "Experimental Control in Software Reliability Certification," *Proceedings of the 7th Annual NASA/Goddard Software Engineering Workshop*, November 1994, reprinted in *Cleanroom Software Engineering: A Reader*, C. J. Trammell and J. H. Poore, eds., Cambridge, Massachusetts: Blackwell, 1996.
- [9] S. Fujiwara, G. v. Bochmann, F. Khendek, M. Amalou, and A. Ghedamsi, "Test Selection Based on Finite State Models," *IEEE Transactions on Software Engineering*, v. 17, n. 6, June 1991, pp. 591-603.