Abstract

A test plan generation algorithm is proposed for systems such as process-control systems or transaction-processing systems. Input to the algorithm is a description of the functional requirements for a given system. This is converted to an augmented finite state automaton (fsa) from which a regular grammar is derived. The grammar is used to generate test "sentences" each of which describes a sequence of stimuli to be applied to the system under test and responses required of the system under test.

Introduction

Testing of software systems is a time-consuming, labor-intensive task. For systems of realistic size and complexity, the generation of an effective set of tests can be a substantial task involving the analysis of thousands of combinations of cases. For large-scale systems, automation of the test plan generation process is the only way one can hope to generate an effective set of tests. Changes in specifications during the development and maintenance phases of the program life cycle only serve to reinforce this position.

Systems which generate only test data solve half the problem. A test case (as opposed to test data) defines not only a set of inputs to the system under test, but also the outputs expected of the system [NEY76]. Given a set of test cases, one can automate their execution (e.g., [PAN76, BAU78]); given only test data this cannot be done because human intervention is required to interpret the results.

This paper addresses one aspect of automating the testing process, namely, function test plan generation. We further restrict the discussion to one class of systems - those for which a particular functional specification technique is well understood. The test plan generation technique described herein is novel in that it uses a regular grammar derived from the functional specification. The scheme not only generates a complete set of input sequences but generates the expected output sequences as well.

We also describe how such a test plan generation scheme fits into a software program development system which includes a table-driven compiler to analyze the functional specifications and an automatic test executor which applies the generated tests and reports on the results.

Other Work in Test Plan Generation

Efforts to automate the process of test data generation have for the most part used the "software-under-test" as the basis for the process. Using this approach, the software to be tested is analyzed and test data is generated to meet certain criteria such as all statements being executed at least once or all branches being traversed at least once. Several disadvantages of this approach can be identified.

As Goodenough and Gerhart [GOO75] have observed, knowledge of a program's internal structure is not sufficient for the generation of a meaningful set of test data. For example, the implementation of a given program feature may have been totally ignored by the
program's implementers. Tests based on the structure of such a program will not uncover the absence of the feature.

More significantly, automated test generation based solely on the program to be tested only does half the job. In the absence of a specification of the software's behavior, one can only hope to generate data to exercise the software. As discussed above, the tester is still faced with the manual task of determining the expected behavior for each case.

One reason most test plan generation schemes are based on the software rather than its specification is that we lack formal techniques needed to specify the behavior for most practical software systems. However, there exist some types of practical systems for which formal specification techniques do exist. In this paper we describe a test plan generation scheme for one such class of systems.

The systems we have in mind are those which can be specified using an augmented finite state automaton (fsa) model [Woo70]. Woods used this model to describe a system which performs natural language analysis. There are many other systems whose functional behavior can be specified using such a model.

The call-processing component of telephone switching systems is often specified in this fashion [KAH71]. An augmented fsa is used to describe the many states of a call sequence where each arc emanating from a state represents a stimulus (e.g., DIALS A DIGIT) and the associated system response(s) (e.g., DIAL TONE IS NO LONGER HEARD).

Another example of a system which can be specified using an augmented fsa model is that of an airline reservation system. For such a system the augmented fsa specifies the states of a reservation. Each arc specifies for a given transaction type, the required system response and the resulting state of the reservation.

Note that in the above examples, the fsa model does not refer to states of the system as a whole; rather it refers to states of a call sequence or airline reservation. There are far too many system states to make modeling feasible (for example, the system state of a telephone switching system must comprise the states of every handset and every trunk). However, a model which highlights observable states of a given function proves to be quite useful.

Given a specification for the observable behavior of a system which can be modeled by an fsa, we wish to generate a set of test sequences to test the system thoroughly. Each test sequence will be a sequence of stimuli (inputs) and the associated responses. Figure 1 illustrates an excerpt of a test sequence for a telephone switching system.

Figure 1

For the remainder of the paper we will restrict our discussion to the testing of telephone switching systems since this is an application representative of fsa-modellable systems. The effective generation of test plans for such systems is a current practical problem for their implementers. However, the techniques we will describe apply to all systems whose observable behavior can be modeled with an augmented fsa. Such systems include many real-time systems, process-control systems, and transaction-processing systems.

Other work in this area of test plan generation has been done at Bell Laboratories [JES76]. Their ATLAS system employs a set of programs called the Enumerator of Loops, Covers, and Admissible Paths (ELCAP). ELCAP operates on a directed graph representation of an augmented fsa. The approach we are suggesting differs in that a grammar-generative technique is used which we believe is simpler to understand and implement. (The constraints on the augmented fsa differ significantly as well).

Other work involving grammar-generative techniques for test data generation include [PAY78] in which test programs are generated for input to a compiler under test. Also, Duncan [DUl78] advocates the manual writing of test grammars not only to aid in the generation of test sequences but also as a documentation technique for tests.
This section describes the context in which the test plan generation scheme operates.

Figure 2 is a block diagram illustrating the components of an automatic test system for telephone switching systems. The test system requires as input a formal specification of the system to be tested. It generates test sequences based on one of several completeness criteria, and finally it applies the test sequences to the system under test and reports on the results. In Figure 2, the test plan generation scheme is implemented in a component called the Test Plan Generator (TPG).

Input to the TPG is an augmented fsa which describes the observable behavior of the system to be tested. The augmented fsa is an fsa with an additional feature -- associated with each arc is a condition which must be satisfied for that arc to be traversed. Like a standard fsa, the augmented fsa has a distinguished initial and final state.

The augmented fsa is generated from a formal prose description of the system by a component called a requirements language processor (RLP) [DAV78, DAV79]. The RLP merges all the information and checks that the specification is consistent. It assures that all states are reachable from a given initial state, that a system response is defined for each state-input combination, and that no conflicting state-input-output triples exist. In addition to producing an augmented fsa representation of the system, the RLP produces a cross-reference listing and a graphical representation of the model. See the example later in this paper for sample input to the RLP.

The purpose of the TPG is to analyze the functional description produced by the RLP and to produce a set of executable test "scripts". Each script defines a sequence of stimuli to be applied to the system under test and responses expected from the system under test.

The test scripts are input to the third component of the test system, an automatic test executor (ATE) [BAU78]. The ATE component executes each test script by applying each stimulus at the appropriate time and verifying, when appropriate, that the system responds in

---

**Language Definition**

Tables for Application

<table>
<thead>
<tr>
<th>Type 2</th>
<th>Augmented fsa</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Model</td>
</tr>
<tr>
<td></td>
<td>Concordance Listing, State-Transition Diagrams</td>
</tr>
</tbody>
</table>

---

Figure 3: RLP Inputs and Outputs

427
the desired manner. A failure report is printed for each test script in which the system fails to respond as described.

The basic components of an ATE test script are the CAUSE and VERIFY statements. The CAUSE statement is used to apply a stimulus to the system under test (e.g., "CAUSE GO_OFF_HOOK (LOCAL_A)"); the VERIFY statement is used to verify that the system responded as intended (e.g., "VERIFY RECEIVING_DIAL TONE (LOCAL_A)"). A macro capability allows one to specify a parameterized sequence of operations with one statement. Facilities also exist for controlling the timing of statement execution, specifying actions to be taken when a system under test failure is detected, and for pseudo-concurrent execution of test scripts.

Like the RLP, the ATE is designed to be used for a large class of systems to be tested. Its interface to a given system under test (SUT) is through a user-provided hardware harness and associated I/O software (Figure 4). Tables in the ATE map application-dependent stimuli and responses to the appropriate I/O operation on the harness.

**The Basic Approach**

Our approach to test plan generation is predicated on a few assumptions about the model as derived from the requirements document. It has been demonstrated in principle that the process of constructing a state transition matrix representation of a finite state machine from a structured requirements document can be automated [DAV78]. Such a requirements language processor can ensure that there are no inconsistencies in the requirements. That is, the document may not specify two different "next states" for a test script current state/input-symbol (i.e., stimulus) pair. This consistency check permits the test plan generation process to assume that the state transition matrix represents a deterministic finite state machine. We can ensure that all states are reachable by computing the transitive closure of the Boolean matrix R which is obtained from the state transition matrix as follows:

R is a square matrix with as many rows as there are states. The (i, j) element of R is set to 1 if state j can be reached by a path of length one from state i, as indicated by the state transition matrix. All other elements of R are set to 0. Now suppose state K is identified as the starting state for a particular set of tests. Then, in order to ensure that every state is reachable from state K, we require that row K in the transitive closure of R be all 1's. This indicates that every state is reachable from state K by a path of finite length.

A result from automata theory [HOP69] describes how to construct a grammar for the regular language accepted by a finite state machine. We need to construct a similar grammar and augment it in two ways. First, since our tests must verify that a particular response was observed after a particular stimulus was applied, the observable outputs from the finite state machine must be terminal symbols in our grammar. Second, the "next user function state" is often dependent on a "system state", as well as the current state/input-symbol pair. This is the
case, for example, when a telephone is lifted off hook, and the system responds with a dial tone only if a digit-collecting register is available. In general, there are many instances where the next state observable by the user is dependent on the availability of some system resource. When this is the case, the set of tests generated must take into account all relevant system information for each state transition. It is useful to view the elements of the state transition matrix as consisting of CASE statements, where the "case-values" are the relevant system information for each user state/input-symbol pair. Now our grammar must be augmented to indicate this choice of next states, and its dependence on the system.

Example

Our example for demonstrating the transformations from finite state machine to a grammar and from the grammar to a test script, will be a local-to-local call. There are four entities involved in describing the machine: states of the call, user inputs, system "states", and system responses. We shall use a different kind of symbol to represent each entity, with the correspondence as indicated in Table 1.

The requirements for a local-to-local call might start with [DAV78]:

```
IN IDLE STATE
WAIT FOR CALLER GOES_OFF_HOOK
IF SYSTEM NOT_BLOCKED
THEN CALLER DIAL_TONE_STARTS
    IN RECEIVE_DIAL_TONE STATE
    WAIT FOR CALLER DIALS_LOCAL_DIGITS
    IF NONBUSY_STATION
    THEN CALLER RINGBACK_STARTS
```

Figure 5 shows the state transition matrix corresponding to a complete specification of a local-to-local call.

This matrix indicates, for example, that when a user function is in the idle state (state I), and goes off hook (input 1), then, if the system is not blocked (system state B), a dial tone will be heard (output a), and a transition to the "receive dial tone" state (state II) occurs.

User Function States (Represented by Roman Numerals)

1. Idle
2. Receive dial tone
3. Receive ringback tone
4. Local to local connect
5. Locked out

User Inputs (Represented by Decimal Integers)

1. Calling party goes off hook
2. Calling party dials local digits
3. Called party goes off hook
4. Calling party goes on hook
5. Called party goes on hook

System "States" (Represented by Capital Letters)

A. System blocked
B. System not blocked
C. Class Restricted
D. Nonbusy station
E. Conventional busy station
F. Busy pilot station / free station found
G. Busy pilot station / no free station found
H. Vacant line

System Outputs (Represented by Small Letters)

a. Dial tone starts
b. Intercept sequence
c. Ringback starts
d. Busy signal starts
e. Ringback stops
f. Connection made
g. Disconnect
h. Initialize

Table 1

429
The grammar we construct from the state transition matrix in Figure 5 is given by the set of productions in Figure 6. The initial state and the final state are both state I, so the non-terminal <I> is the start symbol in the grammar.

The following sequence of sentential forms illustrates the generation of the sentence corresponding to the sequence of events in a successful local-to-local call:

\[<I> ::= A1<I> | A1 | B1A<II>\]
\[<II> ::= C2b<I> | C2b | D2c<III> | E2d<I> | E2d | F2c<III> | G2d<I> | G2d | H2b<I> | H2b\]
\[<III> ::= 3ef<IV> | 4gh<I> | 4gh\]
\[<IV> ::= 4gh<I> | 4gh | 5g<V>\]
\[<V> ::= 4h<I> | 4h\]

Figure 6: Grammar Derived From State Transition Matrix of Figure 5.

The capital letters represent system conditions which must be forced. Small letters are points at which system responses must be verified. Integers are inputs from the user. The test script segment corresponding to the above sentence could be:

```plaintext
(B) FORCE_SYSTEM_NOT_BLOCKED;
(1) CAUSE GO_OFF_HOOK(CALLING_PARTY);
   (a) VERIFY DIAL_TONE_STARTS(CALLING_PARTY);
   (D) FORCE_NONBUSY_STATION(CALLED_PARTY);
(2) CAUSE DIAL_LOCAL_DIGITS(CALLED_PARTY);
   (c) VERIFY_RINGBACK_STARTS(CALLED_PARTY);
   (e) VERIFY_RINGBACK_STOPS(CALLED_PARTY);
   (f) VERIFY_CONNECTION_MADE(CALLING_PARTY,
      CALLED_PARTY);
(3) CAUSE GO_OFF_HOOK(CALLED_PARTY);
   (g) VERIFY_DISCONNECT(CALLED_PARTY, CALLED_PARTY);
(4) CAUSE GO_ON_HOOK(CALLED_PARTY);
   (h) VERIFY_INITIALIZATE;
```

Figure 5
State Transition Matrix for Local-to-Local Call
(Note: Entries are of the form:
\(<\text{system state}>:/<\text{next user state}>/<\text{output(s) to user}>\).
A * in the <system state> position indicates that the state of the system is not interrogated in this case).
Very often the specification of a telephone switch may indicate a time delay between a particular stimulus and its associated response. The language input to the ATE allows for this by including a \texttt{WAIT} statement. It is a simple matter for the RLP to include such information on the arcs of the \texttt{fsa} (i.e., in the entries of the \texttt{STM}), and for the TPG to generate such \texttt{WAIT} statements where appropriate.

So far, the procedure we have described has been quite straightforward, but we have ignored an important consideration. There will be situations where the augmented \texttt{fsa} contains loops, and these would cause the generation process to loop as well. However, we would like some mechanism for terminating the looping but ensuring that we are still generating an "effective" subset of the infinite set of tests.

One criterion that ensures termination is to limit the number of times each production can be used during sentence generation to some constant \texttt{K}. If there are loops in the \texttt{fsa}, then the non-terminals in the grammar that correspond to states within loops can appear several times within a single sentential form. Deleting a rule after it is used \texttt{K} times within the generation of a single test assures termination.

Another criterion for ensuring termination is to identify all the potentially looping non-terminals in the grammar, generate from them all the sentential forms that end with the same non-terminal, and limit the use of these loops to some fixed number of times.

It may be possible to identify some non-terminals as corresponding to states that connect two sub-machines of the \texttt{fsa}. Such is the case for a state, \texttt{X}, when \texttt{X} is not in a loop. Then the test sequences to be generated could be decomposed as well, and two smaller sets of tests generated rather than one combinatorially larger one.

**Conclusion**

By no means do we wish to suggest that the test plans generated by the above scheme be used as the only system tests. Before executing test sequences generated by the TPG, it is expected that module and integration tests will have been performed. Instrumentation techniques such as those suggested by Huang [HUA75] can be used to measure the effectiveness of the TPG-generated test sequences. It is expected that the use of these techniques will often reveal untested code segments. This situation can occur for example as the result of an inadequate specification, or as a result of the existence of defensive code or code which is only executed when the system is heavily loaded. Knowing that certain code segments are not tested should convince the implementers to generate test sequences to augment the TPG-generated test plans. Holthouse has observed [HOL78] that program instrumentation to evaluate test coverage has an additional side-effect. It gives you the opportunity to take advantage of the programmers' "peripheral vision" which often leads to the detection of other program errors.

A TPG using the methods described above can, for a realistically complex system, generate a large number of test cases. Were it not for the existence of an ATE which is capable of rapidly executing a large number of tests, the execution of all these tests might not be cost-effective. Even with an ATE, other techniques such as those suggested by Chow [CH077] should be explored in an attempt to arrive at a more cost-effective test plan generation scheme. The effectiveness of a given scheme will no doubt vary from one system under test to the next.

Factors such as the number of function states, number of stimulus types, and the cyclic nature of the system specification will affect the number of test sequences generated. Further work needs to be done using this test generation scheme to evaluate its effectiveness for testing various systems. To date, we have not had the opportunity to try this scheme on a production system, hence we have no quantitative data on which to report.

The use of an augmented \texttt{fsa} to define the functional characteristics of a system is not new. However, nobody has described the above procedure in which the \texttt{fsa} is converted to a regular grammar from which to generate test "sentences". The approach is not only novel but involves a simple notation and concise recursive algorithm based on a strong theoretical and practical foundation.

**Acknowledgements**

We wish to thank A. M. Davis, W. L. Goodhue, T. J. Miller, B. J. Taylor, and the referees for their many helpful comments and suggestions.
References


