A Method to Automate User Interface Testing Using Variable Finite State Machines

Richard K. Shehady and Daniel P. Siewiorek*
Department of Electrical and Computer Engineering
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

A method has been developed that allows automating a portion of interface testing. A core component of this method is a simple, yet flexible way to specify a formal model of a user interface, named variable finite state machines (VFSM). The model can be converted into an equivalent finite state machine, from which tests can be generated.

The method has been applied to the user interface of Navigator II, a wearable computer system employed by aircraft repair personnel. A VFSM model of the interface was constructed, and used to generate 5,968 tests, each composed of an input sequence and its corresponding expected output sequence. The tests were then applied to an instrumented form of the interface, and the results were compared to the expected output. From the data collected, three error sources were detected in the original interface that had escaped months of previous debugging efforts and field usage.

1. Introduction

Producing error free software has remained a challenging problem for the computer industry, even as the need for correct software increases. In this paper, a method to automate the testing of a user interface is presented. This approach is different from much of the previous work in testing theory, in that it is restricted to the specific case of user interfaces. However, it does borrow from some of the earlier work involving hardware correctness theory, which introduced the concept of the finite state machine (FSM) as a model of a sequential machine.

* Research Supported in part by Boeing Computer Services Grant No. G1151-SJA-94-071

The existing body of software testing can be classified using three major divisions [1]. The most relevant of these for this method are the *dynamic functional* tests, a survey of which is in [2]. Previous work in this area has resulted in several algorithms that can create a test suite from a finite state machine model. Some of the more well known algorithms include the Distinguishing Sequence method [3], the W algorithm [4], Transition Tour [5], the Unique Input-Output method [6], and the partial W method (or W_p method) [7]. The W_p algorithm produces a particularly efficient test suite, and was chosen for use in this paper. Diagnosis for FSMs has also been studied [8, 9].

While FSMs are suitable for many applications, they do not make particularly good models for modern user interfaces. A FSM models a complex interface by using a large number of simple states, with the result that many of the states have little relation to an actual portion of the interface. VFSMs attempt to rectify this situation by employing more sophisticated transition and output functions, enabling them to describe an interface in fewer states than an equivalent FSM. In addition, the increased flexibility allows a state to correspond more closely to an actual part of the original interface. Because VFSM models are both smaller and more intuitive than an equivalent FSM, they can be constructed more accurately, and in less time. VFSMs can be then be automatically converted into an equivalent FSM for computational manipulation, such as test generation.

The method is divided into five distinct steps: model definition, conversion, test generation, test execution, and analysis. In the first phase, a VFSM model is constructed that represents the interface. The conversion phase takes the VFSM as input, and automatically produces a FSM as output. During test generation, a set of input sequences, along with the corresponding expected output sequences, are generated from the FSM.

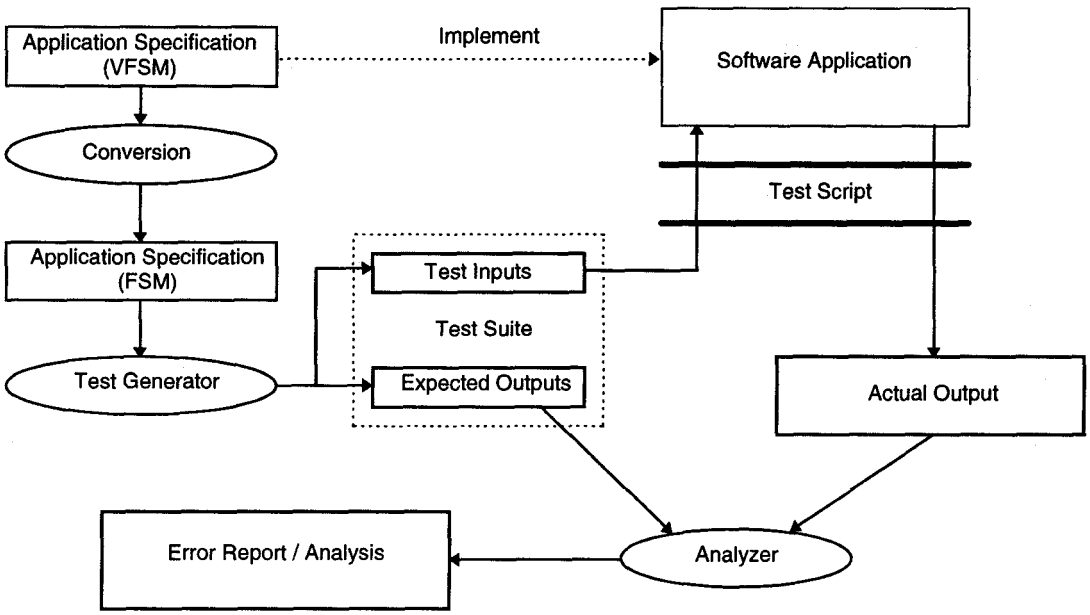


Figure 1. Steps in Testing a User Interface

In the execution step, the test sequences are executed by the interface, which produces a set of actual outputs. Finally, in analysis, the expected output is compared to the actual output. From this information, the errors in the user interface can be deduced. A diagram of this process is presented in Figure 1.

This paper is organized in the following way. Background information is provided in Section 2. The VFSM, and an algorithm to convert a VFSM to a FSM is presented in Section 3. Section 4 discusses the different phases of the method. In Section 5, the results of the application of the method to an existing user interface is detailed. Section 6 summarizes the paper.

2. Background Information

In this section, some relevant information on finite state machines (FSMs) is presented.

A *deterministic finite state machine (FSM)* can be represented as a quintuple M = (S, I, O, T, Φ), where S, I, and O are finite sets of the states, inputs, and outputs, respectively. S_0 represents the initial state. T is the transition function $S \times I \rightarrow S$ that specifies the next state as a function of the current state and the input. Φ is the output function $S \times I \rightarrow O$ that specifies the resulting output from a transition.

A *sequence* of inputs is a series of inputs that is applied sequentially to the machine M. The notation $S_1 \xrightarrow{x/y} S_2$ is used to indicate a transition from state S_1 to state S_2, where x is the input and y is the output. Note that x and y can be sequences. If the output sequence is unimportant, it can be omitted (e.g. $S_1 \xrightarrow{x} S_2$). The notation $S_i | p$ indicates the output generated by applying input sequence p to state S_i.

Two states are *equivalent* if, given any input sequence, they respond with identical output sequences. Two FSMs are equivalent if the initial states S_0 for each FSM are equivalent to each other. A FSM is *fully specified* if, for every element in $S \times I$, there is an output defined in Φ and a transition defined in T.

These concepts can be extended to the VFSM, as shown in the next section.

3. Variable Finite State Machines

It is helpful to describe to VFSM informally before presenting its formal definition.

3.1 Description of a VFSM

A VFSM is similar to a normal FSM, but augmented to allow a number of global variables, each of which can assume a finite number of values during the execution of an input sequence. The value of each variable is used in determining the next state and output in response to an input. Conversely, the value of any vari-

able can be modified by a transition. Variables can be used by the model to specify which transition should be taken for a particular input. A variable can signal to allow or disallow various transitions, depending on its current value.

Effectively, this allows the VFSM to model a state that can use previous inputs as a factor in how to respond to a current input. While the actual interfaces that can be described by a VFSM are the same as for a FSM, a VFSM can do it in far fewer states, resulting in a simpler structure.

3.2 Definition of a VFSM

A n-variable VFSM is defined as a 7-tuple M_V=(S, I, O, T, Φ, V, ς). S, I, and O behave as their counterparts in an FSM.

V is a set of variables, each of which is a set of all of the possible values the variable may be assigned. Thus, V is a set of sets V={V_1, V_2, ..., V_n), where *n* is the number of variables in M_V, and V_i is the set of values that the *i*-th variable may assume.

Both T and Φ must be modified to accommodate the addition of variables. T is now a $(n+2)$ variable function, $D_T \rightarrow S$, where $D_T \subseteq D$ and $D = S \times I \times V_1 \times V_2 \times ... \times V_n$. Similarly, Φ becomes $D_T \rightarrow O$. This is necessary because the current state of each of variables affects both the next state and the output of the VFSM. Note that both T and Φ are functions, and that a VFSM behaves deterministically given any particular input.

The domains of T and Φ are subsets of D because not every combination of variables is possible during execution. If a variable cannot assume a particular value while in a certain state, then that state-variable combination is not necessary in the domain of either T or Φ. An example of this is provided in Section 3.3.

ς is the set of variable transition functions. When a transition is executed, ς is used to determine if any of the variables' values are modified. Like T and Φ, both the current state, the input, and the values of all the variables are needed in the domain of ς. Each variable must also have an initial value, which it possesses at startup.

3.3 Example of a VFSM

The benefit of using a VFSM instead of a FSM can most clearly be seen with an example. Consider a simple user interface with three "screens". The first screen, S_0, is the main menu, where the user can choose to go to either the save screen or the data entry screen. S_1 is the data entry screen, where the user can repeatedly enter data, or return to S_0. S_2 is the save data screen, where the user can either save the data or cancel back to the main menu. The save screen (S_2) is only available if data has already been entered, in S_1. If the user attempts to save the data with no data entered, the interface displays an error message and returns to the main menu (S_0). An intuitive diagram of this interface is presented in Figure 2. Each of the physical screens is represented by one state, and the arcs depict the transitions between them. However, this diagram does not adequately describe the interaction between data entry and the save screen.

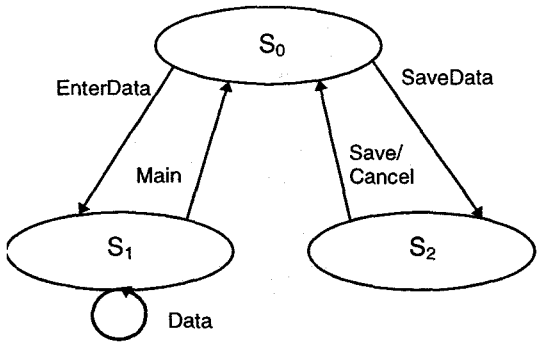


Figure 2. Intuitive Interface Model

A finite state machine of this interface is presented in Figure 3. This model is completely accurate to the actual interface, but it does not promote easy understanding of the interface and how it works. In particular, there are five states in the FSM, while there are only three physical screens. Thus, there is a distinction between an abstract state and a physical screen. In larger interfaces, it becomes increasingly difficult to design and maintain FSM with such unintuitive nuances.

The 1-variable VFSM model of this same interface

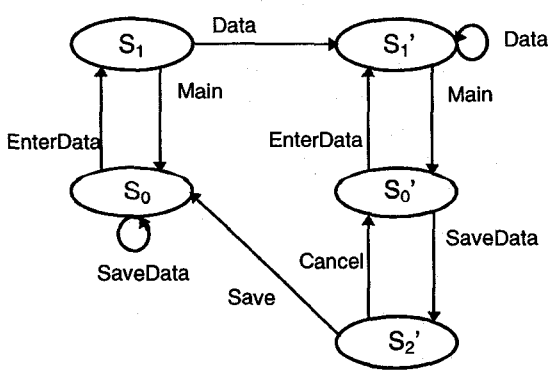


Figure 3. FSM Interface Model

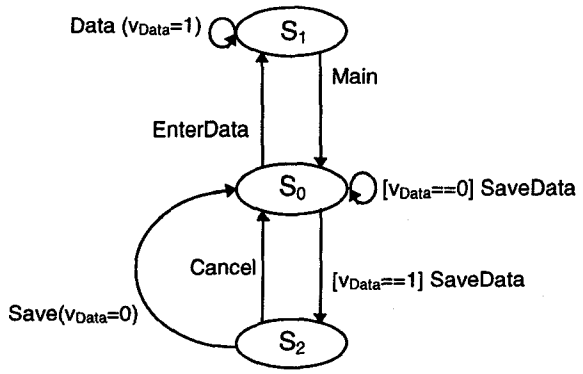


Figure 4. VFSM Interface Model

is shown in Figure 4. The symbol v_{Data} is used to represent the variable, which can possess one of only two values, 0 or 1, at any time. A value of 0 indicates that no new data has been entered, and 1 means that data is currently available. Thus, V = { {0, 1} }. In contrast to the FSM model, there are the same number of states as physical screens. The text associated with each transition arc in the diagram requires some explanation. If there is an expression to the left of the input, this expression indicates the required variable value in order for this transition to occur. An expression to the right of the input indicates that the variable is to be updated during the transition, as directed by the expression.

Both the VFSM and FSM figures completely describe the simple interface. However, the VFSM depiction is much more effective in communicating the function.

This example also highlights the reason why the functions T, Φ, and ς have domains that are subsets of D. Note that the v_{Data} will never have the value 0 while at state S_2. The only transition into S_2 requires that v_{Data} be set to 1, and it is not modified except in a transition that leads out of S_2. Thus, there is no need for the case in which $(S_2, x, 0)$ is input to the transition function, where x is any input.

The method to test a user interface requires that a VFSM model be converted to an equivalent FSM, so that an efficient test set can be generated. Therefore, it is necessary that all possible VFSMs can be converted.

Definition: The *equivalent state set* of a VFSM is $S_{eq} = \{S_i | S_i \in S \times V_1 \times V_2 \times ... \times V_n\}$. Informally, this creates a set of states that combines the information of the states and the variables into one state. In a VFSM, a state must be paired with its current variables to make a transition or output decision.

Theorem I: Every VFSM is equivalent to at least one FSM.

Proof: Let M_V be a n-variable VFSM. Construct a new transition function $T_{eq}: S_{eq} \times I \rightarrow S_{eq}$, that combines the T and ς functions of the VFSM. Note that this is always possible because T and ς have the same domain, and the range of T is S, and the range of ς is $V = \{V_1 \times V_2 \times ... \times V_n\}$, and thus S_{eq} is the Cartesian product of the two ranges. Construct an output function $\Phi_{eq}: S_{eq} \times I \rightarrow O$. Thus, there exists a FSM $M = \{S_{eq}, I, O, T_{eq}, \Phi_{eq}\}$.

The proof of Theorem I demonstrates one way to convert a VFSM into an FSM. A drawback of this method is the presence of a number of unnecessary states in S_{eq}. An alternative is Algorithm 1, which converts a VFSM to a FSM and eliminates the extraneous states. S_{eq}, I, O, T_{eq}, and Φ_{eq} form an equivalent FSM to the VFSM, from Theorem I. If there is a state S_i that is not in the SaveList, then there can be no input sequence x such that $S_0 \xrightarrow{x} S_i$. Therefore, S_i cannot affect the output of any sequence input to S_0, and it can be safely deleted from the machine.

Algorithm 1. Pseudocode for Conversion of a VFSM to FSM

```
Create Seq, Teq, and Feq from VFSM

// Maintain a queue of states to visit
Queue.Insert(S0)

// Maintain a list of states that are
// reachable from S0
SaveList.Insert(S0)

while (queue is not empty)
   s = queue.dequeue()
   for (every transition t in s)
      d = destination state of t
      if (d is not in SaveList)

         // This state is reachable
         // from the start state
         SaveList.Insert(d)
         Queue.Insert(d)

for (every state s in Seq)
   if (s is not in SaveList)

      // These states are not
      // reachable, so delete them
      delete (s)
```

4. Method

As mentioned before, there are five phases in the complete method to automate the testing. These steps are described, in turn.

4.1 Construct a Model of the Interface

From the definition of the VFSM, it may not be obvious that it is a simpler structure than a FSM. With the proper tool, however, most of the complexity can be hidden from the model designer. The T, Φ, and ς functions can all be specified with only a few pieces of information for each state in the VFSM. One way to do this is to list all the transitions for a state. If a variable must have a specific value for the transition, then this variable and its value are marked as required. If a variable is modified during a transition, then the variable and its new value are recorded with the transition. In this way, the specification of a VFSM has changed from listing its entire functions to only marking down the exceptions.

The simple interface from Section 3 serves as a good example of how to specify a VFSM. Listing 1 shows the source code for the VFSM in Figure 4. The `INPUTS`, `OUTPUTS`, `VARIABLES`, and `STATES` sections list the members of the I, O, V, and S states, respectively.

The `TRANSITIONS` section has several different components. Each `@arc` tag indicates that the line describes a transition. The syntax for these lines is:

```
@arc <State> <Input> <NextState> <Output>
```

If this line is not modified by a `@set` or `@req` line, then the transition is assumed to be valid for any variable values, and it will not modify any variables. The `@set` and `@req` have similar formats:

```
@req  <Variable>  <Value Required>

@set  <Variable>  <New Value>
```

In both cases, they modify the transition specified in the previous `@arc` line. A `@req` tag indicates that the transition is only valid if the variable specified has the value required. Multiple variables may be selected by multiple `@req` lines, but any variable not explicitly mentioned is considered valid with any value. It is an error if two transitions have conditions that could both be met by at least one set of variable values. The `@set` instruction assigns the variable the value specified by the `<New Value>` field. Note that in this simple example, only values of 0 and 1 are used. In a real interface, it may be useful to have a much larger set of values. In this case, the `@req` command can be expanded to allow certain ranges of values be acceptable (e.g. $3 < Variable < 10$), and the `@set` command be used to arithmetically modify the variable (e.g. $Variable := Variable + 1$). In this latter case, special considerations are necessary to restrict the variable value to a reasonable finite set, since VFSMs are not able to handle infinite variables. More details on how to specify a VFSM can be found in [10].

4.2 Conversion of VFSM to FSM

The algorithm provided in Section 3 is adequate in converting a VFSM to an FSM in theory. In practice, however, there are a few extra considerations. Both are responses to the test generation algorithm chosen, the W_p method [7].

The W_p method requires that the FSM be fully specified. However, many interfaces respond to a different set of inputs for different states, and it would be tedious and excessively complex to add transitions that simply point back to the state, with NULL outputs. A solution is to have the conversion algorithm automatically add reflexive transitions and NULL outputs wherever needed to fully specify the FSM. In the VFSM specification, these transitions can simply be omitted for

Listing 1. Source Code for a Simple VFSM

```
[INPUTS]
SaveData     Save          Data
EnterData    Cancel        Main

[OUTPUTS]
O0   O1   O2

[VARIABLES]
VData

[STATES]
S0   S1   S2

[TRANSITIONS]
@arc     S0   EnterData     S1   O0
@arc     S0   SaveData      S2   O0
@req     Vdata 1

@arc     S1   Main          S0   O1
@arc     S1   Data          S1   O1
@set     Vdata 1

@arc     S2   Save          S0   O2
@set     Vdata 0
@arc     S2   Cancel        S0   O2
```

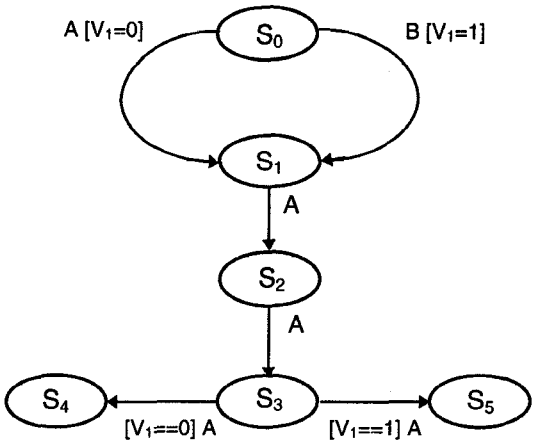


Figure 5. Variable Delay

clarity.

The second consideration is a much more subtle one. The W_p method generates its test sequences, in part, by finding *identification sets.*

Definition: A set of input sequences W_i is an *identification sequence* of state S_i if and only if for every state S_j in S ($i \neq j$), there exists an input sequence p in W_i such that $S_i|p \neq S_j|p$, and that no subset of W_i has this property [7].

The length of the test sequences, as well as the computation time of conversion of VFSM to FSM, depends on the size of the W_i sets. Therefore, it is advantageous to have the smallest W_i sets possible. Consider the VFSM shown in Figure 5. This VFSM represents an arbitrary interface with one variable, V_1. When this VFSM is converted into an equivalent FSM, there will be two versions of states S_1 and S_2; one for V_1=0, and the other for V_1=1. Let $S_1[0]$ represent the state S_1 with V_1=0, and $S_1[1]$ represent the state S_1 with V_1=1. Because there is no dependence on the variable V_1 in any of the transitions for either S_1 or S_2, any input sequence of two elements or less input to $S_1[0]$ or $S_1[1]$ will respond with identical output. Thus, the W_i for both $S_1[0]$ and $S_1[1]$ will contain at least one sequence of three elements.

Definition: The *variable delay* is the minimum path length between the modification of a variable (the destination state of the modifying transition) and a state with a transition that can be affected by that variable.
The variable delay for V_1 in state S_1 is therefore 2.

In larger interfaces, delays can be much higher, adding significantly to the length of the resulting test sequences. Large variable delays increase the length of the sequences in the W_i set for a state.

One technique to avoid large W_i sets, called *output hashing*, is to combine the variable values and the output, similar to how the S_{eq} set was created. Thus, the output set O in the FSM is replaced by $O_{OH} = O \times V_1 \times V_2 \times ... \times V_n$, and the output function Φ_{OH} becomes $\Phi_{OH}: S_{eq} \times I \rightarrow O \times V_1 \times V_2 \times ... \times V_n$. Any two states in S_{eq} that are derived from a single S_i in S will generate different output for any input sequence. Therefore, a variable always changes the output of every transition, which reduces the variable delay to 0. This results in smaller W_i sets (or shorter sequences in W_i), and thus, shorter test sequences.

There is a price to pay for this reduction in potential W_i size. Often, one state in the VFSM corresponds to one section of code in the user interface code. For the output of the interface to match the output of the model, the interface must be aware of which variable state it is in, so that it can add this information to the output. In many cases, this can be performed by maintaining global variables that are used to track the current value of the variables; whenever the interface generates an output, it appends the data from the global variables. This is part of the instrumentation code, describe more fully in Section 4.4.

4.3 Test Set Generation

At this step of the method, a VFSM model of the interface has been created, and converted into a FSM model. For the actual generation of test sequences, the W_p method is used. This algorithm is described and proved in [7]. The key points of the algorithm are 1) it is guaranteed to find any discrepancy between a FSM model and an implementation of the model, based on the outputs of each, and 2) it requires the FSM to be fully specified. Once the test sequences are found, the expected output sequences can be determined by applying the test sequences to the FSM model, and recording the outputs.

Because the W_p algorithm expects the FSM to be fully specified, an interface model may have a large number of `NULL` transitions, which are transitions that arc back to the originating state, with `NULL` outputs. The W_p algorithm does not distinguish between these and other transitions, and this can lead to the generation of a number of test sequences that differ only by a `NULL` transition. In some interfaces, this is worthwhile, because it tests that the interface does not react inappropriately to an input that it should not react to at all. However, this does lead to more and longer test sequences. If it is decided that the drawbacks outweigh

the benefits, or if the interface is designed in such a way that these unexpected inputs are not handled gracefully, then the NULL transitions can be deleted from the test sequences after generation. It should be noted that some of the resulting sequences may be redundant if they differed from another sequence only by the NULL transition. In this case, one of the sequences should be deleted from the test suite.

4.4 Execution of the Test Sequences

The primary concern of this step is the instrumentation of the interface to allow automation of the execution. The interface must be instrumented in such a way as to read the input sequences as actual user inputs, and to record the output so that it can be analyzed later. A method that satisfies both requirements is the replacement of user I/O with file I/O. The interface can be modified to read a sequence of inputs from a specified file, and record its actions to an output file. Altering the interface code always raises the possibility of introducing errors that do not exist in the original interface. This is one reason why the output hashing technique described in Section 4.2 carries some risk; it requires additional instrumentation code in the interface to properly name the output. Ultimately, the input sequences that cause errors during the execution phase should be applied to the original interface, to confirm that the errors are not due to instrumentation code. In addition, it is possible that the instrumentation code will mask other, true errors.

Generally, there will be some control script that executes the interface with a specific test sequence, records the output, stores both in an easily analyzed format, and repeats the process for the next test.

4.5 Analysis of the Data

The primary result of the execution of a test sequence is whether or not the interface's output matches the expected output generated by the model. If so, then the interface is said to have *passed* that particular test. If the interface passes the entire test set, then by the W_p algorithm, there is no discrepancy between the model and the interface.

5. Experimental Interface

The method described above was applied to the Navigator II interface, developed at the Engineering Design and Research Center (EDRC) at Carnegie Mellon University. This interface was designed for aircraft repair personnel equipped with a wearable computer [11]. The Navigator II was chosen as an ideal test vehicle because the interface was complex enough to test the viability of the method, and because it had already undergone significant debugging efforts.

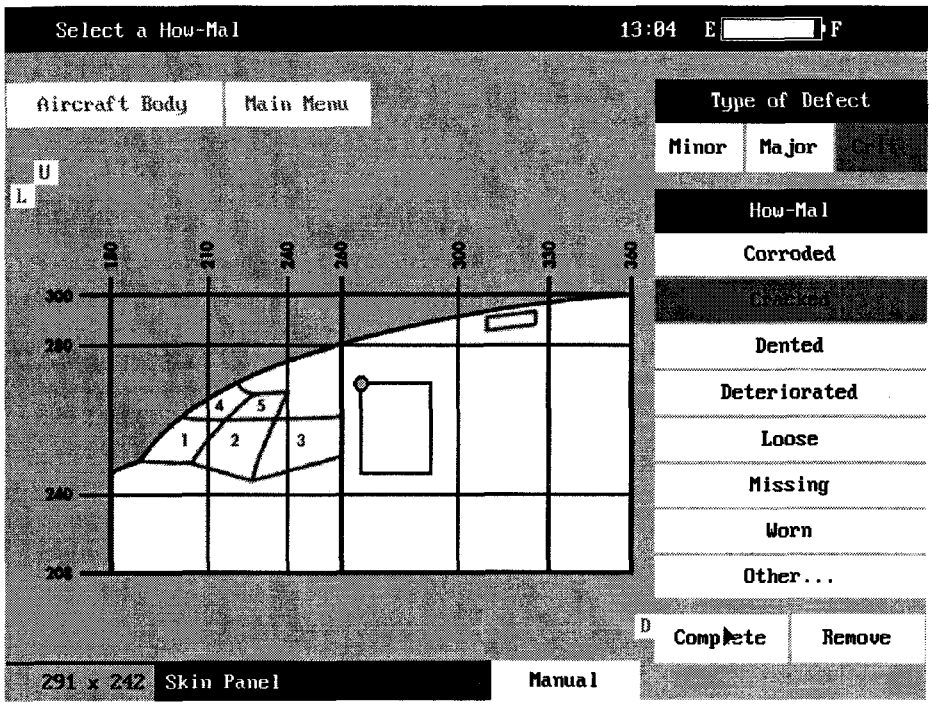
The interface was designed to allow repair personnel to enter the location and types of defects during an inspection of an aircraft. Speech was used as the primary input to allow hands-free usage. The entire interface was comprised of approximately fourteen different screen layouts, with some variations occurring in response to user input. Figure 6 depicts a screen shot of the actual interface.

The model of the interface was a six variable VFSM, containing twenty states, forty-six inputs, and twenty-one outputs. Two of the inputs were not actual user inputs, but rather signaled the interface code that a particular type of input should be simulated. In the Navigator II interface, various airplane parts and the types of defects associated with them are loaded from a database file, and are subsequently used as allowable user input. These type of data are called *data driven input.* Including all of these inputs would have increased the size of the VFSM model dramatically. Further, testing all of these data was not the goal of this method. When the instrumented interface received a signal input, it would generate an appropriate input from the available legal options, as described in [10].

Table 1 shows the relationship between the type of model used and the resulting complexity. The FSM row corresponds to the FSM generated from the VFSM using Algorithm 1 alone. The Output Hash row represents the FSM generated by Algorithm 1, and employing output hashing as described in Section 4.2. The most striking piece of information is that the VFSM only needed 20 states and 93 transitions to completely model the interface; in comparison the FSM required 580 states and 26,680 transitions. This is a clear indication of how a VFSM can reduce the apparent complexity of an interface. Constructing the FSM model by hand would be extremely tedious at best, and most likely error prone as well. By contrast, the 20 states of the VFSM corresponded well with the actual screens of the user interface, and were relatively easy to model.

Table 1. Comparison of Model Types

Type	States	Outputs	Transitions
VFSM	20	20	93
FSM	580	20	26,680
Output Hash	580	640	26,680



There were four different test suites generated, each representing one of the following cases. *Case I) Base Case:* This test set did not employ NULL transition removal or output hashing. It should be expected that this case would generate the largest number of test sequences, and largest average length per input sequence. *Case II: NULL Transition Removal.* In this case, NULL transition removal was used. The average length of input sequences should drop, as well of the total number of test sequences. Some error coverage could be lost as a result of NULL transition removal, however. *Case III: Output Hashing.* Output hashing should result in significantly lower average length input sequences, as well as the total number of sequences. Although maximal coverage should result, this case does require additional instrumentation, as described in Section 4.2. *Case IV: Output Hashing and NULL Transition Removal.* This case should result in the smallest test set of all, but with the corresponding loss of error coverage and increased instrumentation requirements. The tests sets that were generated are compared in Table 2.

Note that by employing output hashing alone, the number of total inputs was reduced by a factor of almost 22 times, from 1,376,085 to 62,687, with no loss of coverage. For the case of Navigator II, instrumenting the code to produce hashed outputs was straightforward, and unlikely to cause errors, and thus was considered a clear advantage. Removing the NULL outputs was almost as effective in reducing the test set size, but it does result in loss of coverage.

Table 2. Comparison of Test Sets

Case	Tests	Total Inputs	Inputs/ Test	NULL Trans.
I	103,063	1,376,085	13.4	258,684
II	6,125	80,889	13.2	0
III	5,698	62,687	11.0	6,226
IV	1,190	12,639	10.6	0

The test sequences from Case III (Output Hashing alone) were selected to be executed on an instrumented version of the interface. The instrumentation required two major modifications, I/O re-rerouting and variable tracking. I/O re-routing involved conversion of the interface from speech-based input and graphical output to a file based system. The input was read in from a file, and the output was written to a separate file. Note that the graphical output was left intact, to minimize the code changes made. The original interface received all of its speech input from a few central functions, making the conversion relatively straightforward. Variable tracking was needed because output hashing was being used, and failure to append the current values of the variables would result in the failure of every test sequence. One global variable was added to the program for every VFSM variable. The values were modified whenever it

was directed by the VFSM model. The output function was then changed to append the values of the variables to the output, which was then written to a file.

This instrumented form of the interface was then executed with the 5,698 test sequences from Case III. The output sequence for each test was recorded, and compared to the expected output. There were a total of 422 failed tests. Upon examination, it was found that all of the failed tests came from five error sources in the instrumented interface. In addition, three of the errors were reproducible on the original, unmodified version of the interface. These errors only manifested after highly specific input sequences were executed. A summary of the error categories is presented in Table 3. The deterministic column refers to whether or not the error would manifest itself in every execution of an input sequence known to cause the error.

Of the two errors that were not reproduced in the original interface, one (C) resulted in an immediate application crash, before even a single input was read. This was never observed in the original interface, nor was it possible to reproduce the crash consistently. Presumably, there was some error in the instrumentation or script code that caused this. The other instrumentation error (D) was the result of a specific set of inputs, but it never manifested in the original interface. Therefore, it must be considered an instrumentation error as well, although it may be that it is a non deterministic error that never manifested itself during a run of the original code.

Table 3. Summary of Errors

	Deterministic	Observed in Original	Number Observed
A	Yes	Yes	303
B	No	Yes	54
C	No	No	53
D	No	No	11
E	Yes	Yes	1
		TOTAL	422

The results are encouraging because the errors that were found only surfaced after highly specific input sequences. All three of the interface errors had escaped months of previous debugging efforts, but were pinpointed quickly after analyzing the test results. The input sequences were not obvious boundary conditions, and thus were unlikely to have been developed in a general debugging session.

6. Summary

A method to automate user interface testing was presented, using the VFSM as the interface model. The VFSM was shown to reduce the complexity of modeling interfaces while retaining the ability to handle sophisticated relationships. An experiment applying the method to the Navigator II interface was presented, with encouraging results in the ability to find errors that escape conventional debugging efforts.

Potential future work consists of investigating methods to automate the diagnosis of errors, and incorporating data driven input into the model. In addition, a graphical tool that assists VFSM model entry would allow easier and less error-prone model construction over the current text description.

References

[1] S. Andriole, "Software Validation, Verification, Testing & Documentation," Petrocelli Books, Princeton, NJ, 1986.

[2] W.E. Howden, "A Survey of Dynamic Analysis Methods," in Tutorial: Software Testing & Validation Techniques, IEEE Press, 1981.

[3] G. Gonenc, "A method for the design of fault detection experiments," IEEE Trans. Computer, Vol. C-19, pp. 551-558, June 1970.

[4] T.S. Chow, "Testing Design Modeled by Finite State Machines," IEEE Trans. on Software Eng. 4(3), 1978.

[5] S. Naito, M. Tsunoyama, "Fault Detection for Sequential Machines by Transition Tours," Proc. of Fault Tolerant Computing Symposium, pp. 238-243, 1981.

[6] K.K. Sabnani, A.T. Dahbura, "A protocol testing procedure," Computer Networks and ISDN System, 15(4): 285-297, 1988.

[7] S. Fujiwara, G. V. Bochmann, F. Khendek, M. Amalou, A. Ghedemsi, "Test selection based on finite state models," IEEE Trans. on Soft Eng., 17(6): 591-603, 1991.

[8] A. Ghedemsi, G. v. Bochmann, "Diagnostic Tests for Finite State Machines," Univ de Montreal, Montreal, January 1992.

[9] A. Ghedemsi, "Test selection and diagnostic methods," TR, Univ de Montreal, Montreal, February 1991.

[10] R. Shehady and D. Siewiorek, "A Methodology to Automate User Interface Testing," EDRC Tech Report, December 1996.

[11] A. Smailagic, D. Siewiorek, "Modalities of Interaction with CMU Wearable Computers," IEEE Personal Communications, 3(1): 15-25, February 1996