# 26

# Validation-Based Test Sequence Generation for Networks of Extended Finite State Machines

**Samuel Huang**
*Lucent Technologies Inc., 2000 N. Naperville Rd., Naperville, IL 60566.*
*E-mail:* shuang@ihgp.lucent.com.

**David Lee**
*Bell Laboratories, 600 Mountain Ave., Murray Hill, NJ 07974.*
*E-mail:* davidlee@bell-labs.com.

**Mark Staskauskas**
*Bell Laboratories, 1000 E. Warrenville Rd., Naperville, IL 60566.*
*E-mail:* markstas@bell-labs.com.

**Abstract**

We describe a method for the automatic generation of test sequences that has been implemented as part of the Virtual Finite State Machine (VFSM) toolset. VFSM has been used extensively in software development for the Lucent Technologies 5ESS® switching system. The method produces a set of sequences of messages which, when provided as input to a VFSM, result in nearly complete code coverage. The method relies on information collected by the VFSM validation tool, which uses Holzmann's supertrace algorithm to explore execution scenarios of a network of VFSMs and search for errors such as deadlock and unexpected inputs. We have applied our method for the generation of test sequences to a VFSM implementation of the ETSI Intelligent Network Application Protocol (INAP). Because of the large size of INAP, we were unable to validate it completely due to state explosion arising from the exhaustive search that the VFSM validator performs. We therefore devised a novel partial validation technique that attempts to generate a subset of the state space of a network of VFSMs and derives test sequences that maximize the code coverage and minimize the number of tests. In this paper, we present an overview of our test-sequence-generation method and its application to the VFSM implementation of INAP.

R. Gotzhein et al. (eds.), *Formal Description Techniques IX*

**Keywords**
Practical experience and case studies (I.2); Verification, validation and testing (I.8); Tools and tool support (I.9); Protocol testing (IV.2); Test case selection and test coverage (IV.4).

# 1 INTRODUCTION

Testing is one of the most expensive and time-consuming phases of the software development process. Part of the reason for this is that little of the testing process is automated. Developers must first devise a set of tests that exercises all portions of the program under test. For each test, the developer must then manually construct the necessary inputs to the program and calculate the expected output of the program for those inputs. Techniques to automate the testing process have therefore been the subject of much recent research.

Over the past four years, we have been involved in the introduction of the Virtual Finite State Machine (VFSM) methodology [Wagner, 1992; Flora-Holmquist et al., 1995a; Flora-Holmquist and Staskauskas, 1996] into the software development organization for the No. 5 Electronic Switching System (5ESS®), a telephone switch that is a product of Lucent Technologies Inc. (formerly the systems and technology unit of AT&T). VFSM allows the control behavior of a software module to be specified at a high level as an extended finite state machine, and enforces a clear separation between the control- and data-related aspects of an implementation. One of the main reasons for the successful introduction of VFSM is that it automates many tasks that developers must perform. For example, the control portion of a VFSM-based implementation is generated automatically from the VFSM specification.

To further enhance the effectiveness of VFSM, we therefore became interested in other ways of automating parts of the VFSM design process. Testing of VFSMs emerged as an obvious candidate for automation. The VFSM toolset includes an interactive simulator that allows the developer to test a VFSM by providing it with inputs, executing it, and observing that the state transitions and outputs produced are in accord with his expectations. The simulator includes a coverage monitor that records which statements of the VFSM are executed during a simulation session, where a statement is either a state transition or the production of an output. It is obviously desirable for a simulation session to include the execution of every statement at least once. Construction of a small set of input sequences that achieves such complete coverage requires a great deal of cleverness and manual effort. Such a set of test sequences could greatly enhance the effectiveness of a VFSM simulation session, since they allow the developer to quickly observe the behavior of a VFSM over a wide range of possible scenarios. These test sequences could also be used later during the testing phase to thoroughly exercise the control structure of the implementation running on an actual 5ESS switch.

The VFSM validation tool [Flora-Holmquist and Staskauskas, 1995b] offers a means of generating such a set of sequences automatically. The validator produces scenarios by enumerating all possible interleavings of executions of the individual VFSMs in a network of communicating VFSMs, using executable representations of the VFSMs that are generated by the VFSM compiler. It is therefore possible to record the statements that a VFSM executes in each step of a scenario. We describe in this paper an algorithm for generating VFSM test sequences that makes use of this information about VFSM statement executions. The algorithm proceeds in two phases. In the first phase, one identifies a "target" VFSM in a network of VFSMs for which test sequences are to be generated and then performs validation on the VFSM network. During

validation, we construct a graph in which there is an edge for each execution step of a VFSM that is labeled with a message it received and the statements it executed upon processing the message. In the second phase, we perform a search on this graph to find a set of paths that is as small as possible with the property that every executed statement of the target VFSM appears in the label of at least one edge of one path. The message sequences and some additional information for the target VFSM labeling the edges of this set of paths therefore guarantee coverage of all statements that were executed during the validation run.

We chose the VFSM implementation of the Intelligent Network Application Protocol (INAP) [ETSI, 1994] as the initial application of our test-sequence generation algorithm. INAP is an application-layer, telephony-specific protocol developed by the European Telecommunications Standards Institute (ETSI) for the exchange of information between a telephone switch like the 5ESS and computing equipment of service providers that implements advanced telephony features. INAP is one of the largest VFSM applications we have encountered and, for reasons elaborated below, the combinatorial explosion we observed when attempting to validate it was especially severe. We therefore devised a novel partial validation strategy that explores only a portion of possible execution scenarios by randomly truncating some of them. The truncation strategy is "greedy" in the sense that it attempts to maximize the number of statements of the target VFSM that are executed. We show that this strategy allows us to quickly obtain a validation run in which roughly 90% of the statements of INAP are executed.

## 1.1 Related Work

As a protocol is specified and implemented, testing is conducted at different stages. There are lower level tests with a variety of names such as deliverable or developer's test. There is high level testing such as *feature* testing. We shall be focused on feature testing based on a reachability analysis from validation. A closely related area is protocol conformance testing, which has been rather active in recent years. To ensure that protocol systems communicate reliably their implementations must be tested for conformance to the specifications. Typically, a system is tested by an external tester, who applies a selected input sequence to the system and verifies that the corresponding output sequence is that which is expected. It is impossible to test all possible input sequences; we want to perform a minimal number of tests with a desirable fault coverage.

A protocol specification is typically broken into control and data portions where the control portion is modeled by an ordinary finite state machine (FSM) [Lee and Yannakakis, 1996a]. Most of the formal work on conformance testing addresses the problem of testing the control portion [Aho et al., 1991; Chanson and Zhu, 1993; Fujiwara et al., 1991; Lee and Yannakakis, 1996a; Sidhu and Leung, 1989; Sabnani and Dahbura, 1988; Yannakakis and Lee, 1995]. Machines that arise in this way usually have a relatively small number of states (from one to a few dozen), but a large number of different inputs and outputs (50 to 100 or more). For example, the IEEE 802.2 Logical Link Control Protocol (LLC) [ANSI] has 14 states, 48 inputs (even without counting parameter values) and 65 outputs. Clearly, there is an enormous number of machines with that many states, inputs, and outputs, and, consequently, brute force testing is infeasible. A number of methods have been proposed which work for special cases (such as, when there is a distinguishing sequence or a reliable reset capability), or are generally applicable but may not provide a complete fault coverage. For instance, there are the D-method based on distinguishing sequences [Hennie, 1964] the U-method based on UIO sequences [Aho et al., 1991; Sabnani

and Dahbura, 1988] the W-method based on characterization sets [Chow, 1978] and the T-method based on transition tours [Naito and Tsunoyama, 1981]. A general polynomial time test generation procedure was reported recently in [Yannakakis and Lee, 1995]. For a survey, see [Lee and Yannakakis, 1996a; Sidhu and Leung, 1989].

The goal of these techniques is to test whether an implementation FSM conforms (is isomorphic) to the specification FSM. However, for EFSM, which typically is a compact representation of an equivalent FSM with hundreds of millions of states, testing for isomorphism is an impossible task and heuristic procedures have been proposed, such as random walk [West, 1986] and guided random walks [Lee, Sabnani, Kristol, and Paul, 1996].

Our goal for testing INAP, which is also EFSM, is different. We do not test for isomorphism of the corresponding FSM's. Instead, we have a clearly specified criterion by the system requirements: each line of VFSM specification code has to be tested at least once. Equivalently, we want to generate a set of tests such that each transition in the original EFSM is tested at least once. It is often called "white-box" testing in system development and analysis. It can be shown [Lee and Yannakakis, 1996b] that this problem is in general PSPACE-complete. However, based on the validation process, we attempt to generate a subgraph, which is significantly smaller than the reachability graph, yet contains all the VFSM specification code. From this subgraph, we apply an optimization procedure to generate test sequences that cover all the specification code with a minimal number of tests. This is not a heuristic procedure, since the code coverage is the goal and can be easily verified. We also note that conformance testing *per se* did not seem particularly useful in the context of VFSM: because the implementation is generated directly from the VFSM specification, a conformance test would merely test the correctness of the VFSM compiler, and extensive usage of VFSM in the 5ESS environment has given us great confidence that the compiler is correct.

There are a number of commercial toolsets that support the generation of test cases for extended finite state machines. TTCN Link [Wiles et al., 1993; Telelogic, 1996] combines Telelogic's SDT and ITEX tools to enable the semi-automated construction of tests specified in the TTCN notation for SDL processes. However, TTCN Link primarily supports the creation and execution of individual tests and does not attempt to automatically produce a *set* of tests meeting a particular coverage criterion. TestMaster [Teradyne, 1996] is perhaps the tool most closely related to ours; it operates by executing an EFSM systematically for all possible combinations of inputs to obtain a set of tests that covers all its transitions or paths. This approach leads to state explosion for large examples, so the user can specify "path constraints" to limit the number of paths considered by the tool. TestMaster determines the possible inputs to the EFSM by analyzing its structure. Our method, by contrast, operates on a network of communicating VFSMs, and the inputs to the target VFSM are generated by executing the VFSMs representing its environment. Early in our work, we attempted to generate tests by executing the INAP VFSM in isolation in a manner similar to that employed by TestMaster; however, most of the tests produced in this way represented behavior that could never be exhibited by INAP's environment VFSMs. Because of its complexity, we found it easier to correctly capture the behavior of the INAP environment with VFSMs instead of with something similar to the path constraints used by TestMaster.

The remainder of this paper is organized as follows. Section 2 presents a brief overview of the INAP protocol. Section 3 describes the VFSM methodology and its use in the 5ESS development environment. In Section 4, we give a precise description of how the VFSM validator explores execution scenarios of a VFSM network. VFSM test sequences and their relationship to execution

scenarios are defined in Section 5. Section 6 sets forth the algorithm for finding a small set of test sequences that obtains complete coverage of the executed statements of a validation run. In Section 7, we describe the partial-validation technique we used to generate a subset of the scenarios when validating the INAP example. Section 8 contains an example of the test sequences we produced; since the work described here is still very much in progress, we also describe some of the shortcomings of the present algorithm. We conclude in Section 9 with a discussion of the further effort needed to develop a robust test-sequence-generation capability in the VFSM toolset.

## 2 INAP OVERVIEW

In this section, we present a high-level overview of the INAP protocol. The description presented here is incomplete; our goal is to touch on a few aspects of INAP that are relevant to our difficulties in validating it. See [ETSI, 1994] for a more complete description.

INAP is a part of the Intelligent Network Capability Set 1 (IN-CS1) architecture, a set of standards that allows the interconnection of computer and communications equipment of different vendors to provide a network offering advanced telephony features. INAP is a protocol for the exchange of data between a Service Switching Function (SSF), which is a telephone switch like the 5ESS, and a Service Control Function (SCF), which is a computer implementing advanced network services. INAP defines a set of commands that allow the SSF and SCF to interact; for example, the SCF can instruct the SSF to play an announcement to the caller and prompt him to enter digits to specify a service request, and the SSF can return the digits entered to the SCF. There are also commands to enable the SSF to send data regarding charging for advanced services to the SCF, and to allow the SCF to instruct the SSF to exercise flow control to limit the rate of incoming service requests.

One possible call scenario that involves INAP is as follows. When a subscriber connected to an SSF wishes to establish a call, an instance of a Basic Call State Model (BCSM) [ITU, 1993] is created to handle the call. A traditional call is completely dealt with by the BCSM instance. The BCSM defines a set of *trigger detection points* (TDPs), events that indicate that the user has invoked an advanced calling feature. When BCSM encounters a TDP, it creates an instance of INAP to mediate interaction with the SCF. The interaction between INAP and SCF includes a series of *event detection points* (EDPs), which are events that may require INAP to inform BCSM of their occurrence. Each EDP can be either "armed" or "unarmed" for a given service type depending on whether or not notification of BCSM is required; this allows the protocol to be tailored to a large variety of services. The INAP instance created for a call is terminated when interaction with the SCF is no longer required.

## 3 VFSM OVERVIEW

The VFSM methodology [Wagner, 1992; Flora-Holmquist et al., 1995a] consists of a *design paradigm*, in which the control behavior of a software module is specified as a finite-state machine; and an *implementation paradigm*, which consists of a design structure that defines the interface between the control specification and the rest of the implementation. The VFSM toolset translates the VFSM specification into executable form and produces templates for the

```
S_Call_Setup {
   IA: I_Call_Setup     ? O_Check_Bill_Status;
       I_Bill_Paid      ? O_Send_Setup_Msg;
       I_Bill_Not_Paid  ? O_Cancel_Request;

   NS: I_Bill_Paid      > S_Wait_For_Reply;
       I_Bill_Not_Paid  > S_Terminate;
}
```

**Figure 1** Example State of a VFSM Specification

modules that interface with the control portion of the implementation. The VFSM methodology has been used on over 75 5ESS software projects, and has been taught to several hundred 5ESS developers.

A VFSM specification is written in terms of states, virtual inputs and virtual outputs. The term "virtual" means that VFSM inputs and outputs are abstract names local to the VFSM: virtual inputs represent conditions in the environment that influence the control behavior of the specified system, and virtual outputs stand for actions to be taken by the system at various points during its execution. The exact binding between these abstract inputs and outputs and their concrete realizations in the implementation is specified by the VFSM implementation paradigm.

A major difference between a VFSM and a traditional FSM is in how inputs are handled. When a VFSM receives an input, it is stored in a set called the Virtual Input Register (VIR), and remains there until it is explicitly removed. VFSM state transitions and the production of virtual outputs can be conditioned on the presence of particular subsets of inputs in the VIR. VFSM is therefore an *extended* FSM notation: the "state" of a VFSM at any point is given by its VFSM state and the contents of its VIR.

Figure 1 shows an example specification of one state of a VFSM. The example illustrates a protocol for setting up a telephone call. The input-action (`IA:`) section specifies that upon receiving a request to set up a call, denoted by the presence of virtual input `I_Call_Setup` in the VIR, the VFSM produces virtual output `O_Check_Bill_Status`. This output checks to see if the calling customer has paid his bill and updates the VIR in a manner described below with the outcome of the check. If the customer's bill has been paid (`I_Bill_Paid`), the VFSM sends a setup message to the switch servicing the called party (`O_Send_Setup_Msg`); if not (`I_Bill_Not_Paid`), the call-setup request is cancelled (`O_Cancel_Request`). The next-state transition (`NS:`) section specifies the VFSM state to be entered next. If the customer has not paid his bill, the protocol enters a terminal state (`S_Terminate`); otherwise, a transition is made to state `S_Wait_For_Reply` to await the response to the message sent to the called party's switch.

Figure 2 shows the structure of a VFSM implementation. The input mapper and output functions provide a "firewall" that enforces the separation of the top-level control behavior defined by the VFSM specification from the low-level data manipulations and functions of the system. As shown to the left of Figure 2, the input mapper receives *events*, such as messages, interrupts and timer expirations, from the environment of the system. Based on the event received and the values of local data structures, the input mapper determines which virtual inputs must

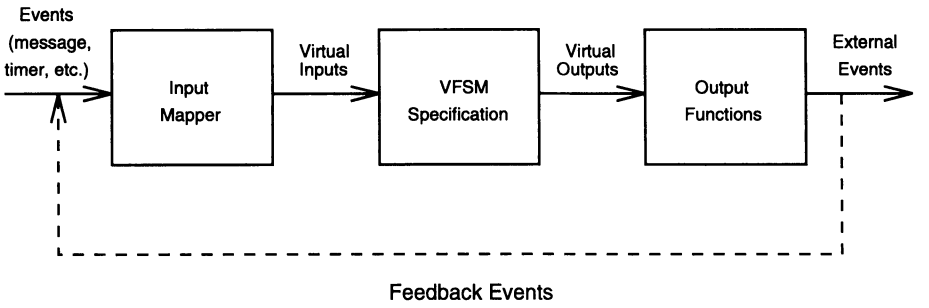



**Figure 2** Structure of a VFSM Implementation

be inserted into, or deleted from, the VIR. When the input mapper completes, the VFSM specification is executed. The VFSM may change state several times and produce several virtual outputs, terminating execution when a state is reached from which no state transition is possible given the current VIR contents. The user associates with each virtual output the name of an *output function*; whenever that output is produced during VFSM execution, its output function is invoked. The output function performs whatever processing and data manipulation are necessary to realize the abstract behavior represented by the virtual output. Note that an output function may also invoke the input mapper with a *feedback event*, as suggested in Figure 2; thus, the VIR can change while the VFSM is executing. A feedback event is used in the example of Figure 1 to allow the virtual output `O_Check_Bill_Status` to update the VIR with the result of checking the caller's bill status.

## 4 FORMAL VALIDATION OF VFSM NETWORKS

Formal validation algorithms search for errors in a system of communicating processes by constructing the *global state graph* of the system. The global state is a vector that contains the state of each process in the system, which includes all information relevant to its communication behavior; the state of a VFSM, for example, includes its VFSM state, VIR and a message queue. Beginning from the initial global state, the validation algorithm generates possible successors of each global state by executing steps of each process in the system. If the size of the message queues is bounded, the number of possible global states is finite, so the generation of global states in this way eventually terminates: it is not necessary to explore successors of a global state that was generated at an earlier point in the validation.

The VFSM validator provides a feature known as *mapping abstractions* that allows the user to specify those aspects of the input mapper and output functions that have an impact on inter-process communication and must therefore be included in the validation model. When a VFSM receives an event, mapping abstractions representing the input mapper allow the user to define all possible combinations of virtual inputs that might be inserted into, or deleted from, the VIR. If more than one combination is possible, the validator will explore a separate successor state for each combination; thus, the mapping abstractions permit nondeterminism. Similarly, the mapping abstraction for each virtual output allows the specification of all possible combinations

**C code in input mapper:**

```
case E_Check_Bill_Status:
    if (custDB->BillStatus == PAID)
        VIR_insert(I_Bill_Paid);
    else VIR_insert(I_Bill_Not_Paid);
```

**Mapping Abstraction:**

```
$name E_Check_Bill_Status
$inlist I_Bill_Paid ||
        I_Bill_Not_Paid
```

**Figure 3** VFSM Validator Mapping Abstraction

```
1  validate(s)
2  {
3      for (each VFSM v for which s(v).queue is non-empty) {
4          for (each c in choices(s(v))) {
5              s' = VFSM_execute(v, s, c);
6              if (global state s' has not been seen before)
7                  validate(s');
8          }
9      }
10 }
```

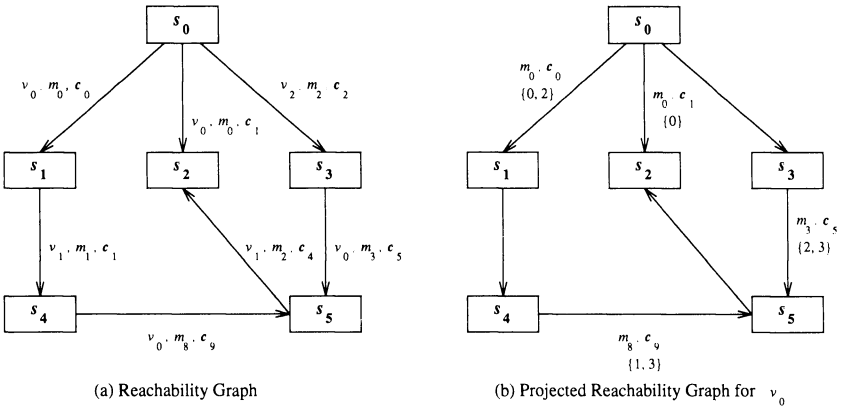
**Figure 4** VFSM Validation Algorithm

of inter-VFSM or feedback events that might be generated by its output function when that virtual output is produced.

Figure 3 presents an example that illustrates the use of mapping abstractions for the VFSM of Figure 1. The example contains a fragment of C code that might appear in the input mapper for the feedback event produced by the output `O_Check_Bill_Status`, along with a mapping abstraction intended to model its behavior. When the event is produced, the input mapper does a database lookup to determine whether or not the caller has paid his bill, and inserts into the VIR either input `I_Bill_Paid` or `I_Bill_Not_Paid` based on the outcome of the lookup. The mapping abstraction for this event simply specifies that either input is possible.

The representation of VFSMs used in the validator is nondeterministic: during its execution, a VFSM may interact with its environment in a way that yields multiple outcomes. We refer to each such interaction that involves a nondeterministic choice as a *choice point*. For each choice point with $n$ outcomes, we assign an integer between 0 and $n - 1$ to each possible choice. A *choice vector*, denoted $\mathbf{c}$, represents one possible assignment of values to the choice points that a VFSM encounters in an execution step. A choice vector therefore completely resolves the nondeterminism of a VFSM execution step: the combination of VFSM state, VIR, received message and choice vector uniquely determines the behavior of the VFSM.

Figure 4 represents a simplified version of the VFSM validation algorithm. The recursive function $validate()$ takes as an argument a global state $s$ of the VFSM network. For each VFSM $v$, the global state contains a component $s(v)$ that consists of its VFSM state ($s(v).state$), VIR




(a) Reachability Graph

(b) Projected Reachability Graph for $v_0$

**Figure 5** Reachability Graph Construction

$(s(v).VIR)$, and message queue $(s(v).queue)$. The algorithm begins by considering each VFSM that is enabled to execute in state $s$ because it has at least one message to process in its queue. For each enabled VFSM, the function $choices(s(v))$ returns the set of possible choice vectors for $v$ given its current state and VIR values and the message at the head of its queue. For each choice vector, the function $VFSM\_execute$ implements the processing of a message by the VFSM: it removes the message at the head of $s(v).queue$, invokes the input mapper, and then executes the VFSM, which may result in a new state and VIR for $v$ and append messages to the queues of other VFSMs, ending in a new global state $s'$. A hashing technique described below is then used to see if $s'$ has been visited before; if not, a recursive call of $validate()$ for the new global state $s'$ is made.

The progress of the algorithm can be understood in terms of the *reachability graph* of the system, an example of which is shown in Figure 5(a). Each node of the graph is a global state, and there is an edge of the graph representing the execution of a VFSM for a particular message and choice vector, as illustrated by the edge labels. Note that there is a node of the graph for each recursive call of $validate()$, and that the algorithm of Figure 4 explores the graph in depth-first order.

## 5 VFSM TEST SEQUENCES

An *execution sequence* $e$ of a VFSM network is an alternating sequence of global states and triples of the form $(v_i, m_i, \mathbf{c}_i)$, i.e. $s_0, (v_0, m_0, \mathbf{c}_0), s_1, (v_1, m_1, \mathbf{c}_1), \ldots$, where, for $i \geq 0$, VFSM $v_i$ executes in state $s_i$ with message $m_i$ and choice vector $\mathbf{c}_i$ as inputs, and the global state that results is $s_{i+1}$. We assume that $m_i$ is the message at the head of $s_i(v).queue$. It is easy to see that the execution sequences of a VFSM network are precisely the paths in the reachability graph of Figure 5(a). A *test sequence* of a particular VFSM $v$ in the network is obtained by projecting the inputs of $v$ from an execution sequence and removing the first element of each input triple. The test sequences of a VFSM can be obtained by constructing the *projected reachability graph* of

the network containing it. Figure 5(b) shows the projected reachability graph for the reachability graph of Figure 5(a), assuming that VFSM $v_0$ is the target VFSM for which test sequences are to be generated. Note that the edges corresponding to executions of VFSMs other than $v_0$ are unlabeled.

As mentioned earlier, our goal is a set of test sequences that causes every statement of the target VFSM to execute at least once. During validation, we can record the statements executed by the target VFSM and add this information to the label of each edge in the projected reachability graph; this information is given in the example in Figure 5(b). Assuming that $v_0$ has four statements numbered 0, 1, 2 and 3, the single test sequence $(m_0, \boldsymbol{c}_0)$, $(m_8, \boldsymbol{c}_9)$ obtains complete code coverage.

The VFSM validator does not normally construct the reachability graph of a VFSM network, since it uses the supertrace hashing technique [Holzmann, 1991], which avoids explicit storage of global states (this permits larger examples to be validated, at the cost of possibly missing some errors). To enable the generation of test sequences, we modified the validator to construct a projected reachability graph like that in Figure 5(b). Note that we do not need to store the entire global state with each node in the graph, since we are only interested in reflecting the correct graph connectivity. The projected reachability graph is the input to the second phase of our technique, which we describe in the next section.

## 6 GENERATING A MINIMAL SET OF TEST SEQUENCES

From the projected reachability graph of the network we want to construct a set of tests such that every statement of the target VFSM is executed at least once. Meanwhile, since each test is costly, we also want to minimize the number of tests.

For clarity, we assign a distinct color to each statement of the target VFSM and the test generation problem can be reduced to a graph color covering problem as follows.

### 6.1 Problem Formulation

Formally, we have a directed graph $G = < V, E >$ with $n = |V|$ nodes, $m = |E|$ edges, a *source* node $s$ of in-degree 0, and a *sink* node $t$ of out-degree 0. * All edges are reachable from the source node and the sink node is reachable from all edges. This is a typical projected reachability graph from the specification VFSM. There is a set $C$ of $k = |C|$ distinct *colors*. Each color corresponds to a statement in the VFSM specification. Each node and edge of graph $G$ is associated with a subset of colors from $C$. † A path from the source to sink is called a *test*.

We are interested in a set of tests that cover all the colors; they are not necessarily the conventional covering paths that cover all the edges. Formally, a *complete test set* covers all the colors in $C$. The path (test) length makes little difference and we are interested in minimizing the number of paths. We shrink each strongly connected component into a node, which contains all the colors of the nodes and edges in the component. The problem then is reduced to that on a

*In the reachability graph there is a number of sink nodes. For clarity, we add a unique sink node $t$ and add an edge from each existing sink node to $t$ so that $t$ becomes the unique sink node in the graph.

†Each statement in the VFSM specification corresponds to a distinct color in $C$ and may have multiple appearances in $E$. We consider a more general case here; each node and edge have a set of colors from $C$.

directed acyclic graph (DAG) [Cormen et al., 1990]. From now on, unless otherwise stated, we assume that the graph $G = < V, E >$ is a DAG.

## 6.2 Algorithm

We need a complete test set - a set of paths from the initial node to the sink node that cover all the colors. On the other hand, each test is costly to experiment and we want to minimize the number of tests: Find a complete test set of minimal cardinality. The problem is NP-hard [Lee and Yannakakis, 1996b].

We consider the following greedy method instead. We first find a test that covers a maximal number of colors and delete the covered colors from $C$. We then repeat the same process until all the colors have been covered. Now the problem becomes: Find a test of a maximal number of colors. The problem is again NP-hard [Lee and Yannakakis, 1996b].

We now discuss a heuristic procedure for finding a test of a maximal number of colors. We topologically sort a DAG [Cormen et al., 1990] such that for each edge $(u, v)$, $u$ has a smaller topological number. We denote the topological number of a node $u$ by $\tau(u)$ and the color set associated with a node $u$ (edge $(u, v)$) by $C(u)$ ($C(u, v)$). We also denote the number of colors in a color set by $|C(\cdot)|$. Obviously, if we conduct a topological sort from the source node $s$ then $\tau(s) = 1$ and $\tau(t) = n$.

The algorithm is in Figure 6. We process the nodes in a reversed topological ordering. For node $v_i$ (of a topological order $i$), we choose an outgoing edge for a path from that node to the sink node that is supposed to cover a maximal number of colors. Specifically, we consider all the outgoing edges from $v_i$: $(v_i, v_j)$ where $j > i$ and $v_j$ has been processed. We compute the number of colors covered by the path from $v_i$ to the sink if we follow $(v_i, v_j)$ and then the computed path from $v_j$ to the sink. We compare all the outgoing edges from $v_i$ and choose the one with a maximal number of colors covered and record that edge $(v_i, v_i')$ at node $v_i$.

This procedure is well defined since $G$ is a DAG. However, it may not provide a maximal color coverage test; when we choose the outgoing edge from $u$, we do not incorporate information of the colors from the source to $u$. However, it works remarkably well for our test generation. See Section 8.

When we process a node $u$ we examine all the outgoing edges $(u, v)$ from the node. For each edge, we take the union of three color sets of no more than $k$ colors each: that of node $u$, edge $(u, v)$, and end node $v$. Therefore, it takes time $O(k)$ to process each edge and the total time and space complexity of the algorithm is $O(km)$ where $k$ is the number of colors and $m$ the number of edges in the graph.

# 7 PARTIAL VALIDATION

The technique for test sequence generation described above assumes that the network of VFSMs to which it is applied can be completely validated. In the case of INAP, however, this was not possible due to its large size. One source of state explosion is the large number of nondeterministic choices often associated with the execution of the INAP VFSM. One example is the selection of event detection points (EDPs) for a given call. Each EDP can be either unarmed or armed in one of two ways; since there can be as many as 13 EDPs to be selected, there are $3^{13}$, or over 1.5 million, possible selections. When combined with the nondeterministic choices of other VFSMs

```
1  maxcolor(s)
2  {
3      topologically sort G from s: v_1 = s, ..., v_n = t;
4      for (i = n, ..., 1) {
5          C = ∅;
6          for (each edge from v_i: (v_i, v_j)) /* j > i */ {
7              C_j = C(v_i) ∪ C(v_i, v_j) ∪ C(v_j);
8              if (|C_j| > |C|) {
9                  C = C_j;
10                 v_i' = v_j;
11             }
12         }
13         C(v_i) = C;
14         associate v_i' with v_i; /* outgoing edge (v_i, v_i') */
15     }
16 }
```

**Figure 6** Algorithm for Finding Maximal Color Set Paths

in an execution sequence, the number of possible sequences becomes truly astronomical; we refer to this phenomenon as *choice explosion*. Another reason for state explosion is the large number of different message types defined by the INAP protocol: there are over 30 different messages that can be exchanged between INAP and the SCF, and over 20 between INAP and the BCSM instance for the call. Because the interactions of INAP with the SCF and BCSM instance can consist of essentially arbitrary sequences of these messages, the number of possible scenarios that can arise is enormous.

We tried several approaches to obtaining a validation run that collected sufficient information for test-sequence generation. The most successful was an approach that randomly prunes depth-first-search paths in a way that achieves better coverage than the traditional algorithm given in Figure 4. Instead of exploring all successors $s'$ of the global state $s$ in the algorithm of Figure 4, we define a pruning probability and draw a random number between 0 and 1 prior to each recursive call of the procedure $validate()$. If the random number is less than the pruning probability, then the recursive call is skipped. However, if the call of $VFSM\_execute()$ that produces global state $s'$ causes some statement of the VFSM to be executed for the first time, then the recursive call is *not* skipped. Thus, the pruning strategy is "greedy" in that it tries to maximize the number of statements executed. Also, since each VFSM execution step that is not pruned causes an edge to be added to the projected reachability graph, the strategy guarantees that every executed statement of the target VFSM appears on at least one edge label in the graph.

We have found that, instead of using a fixed pruning probability, better results are obtained by increasing the likelihood of pruning as the depth of the search, i.e. the distance from the initial node of the graph, increases. This is because skipping a recursive call near the root of the graph causes a larger number of scenarios to be discarded from consideration than skipping a call deeper in the search. For a search depth of $d$, the probability of pruning that we used is given by

| $a_s$ | $a_c$ | Edges | Coverage | Time |
|---|---|---|---|---|
| .18 | .018 | 5 189 | 53.3% | 2s |
| .16 | .016 | 46 347 | 58.9% | 20s |
| .15 | .015 | 288 504 | 89.6% | 2m09s |
| .12 | .012 | 1 363 553 | 87.5% | 9m13s |
| .11 | .011 | 7 309 544 | 92.4% | 50m38s |
| .00 | .000 | 172 129 468 | 79.0% | >14h |

**Table 1** Results of Partial Validation

$1 - e^{-a_s d}$, where $a_s$ is a parameter that can be varied to control the degree of pruning. To deal effectively with the choice-explosion problem, we implemented a similar strategy for pruning possible choice-vector values. The validator generates choice vectors in lexicographic order by "incrementing" the least-significant (last) element of the choice vector and propagating carries leftward. We implemented pruning by randomly skipping values of elements when incrementing them. Since the number of choice-vector values pruned in this way depends on the significance of the element whose value is skipped, we used an exponentially distributed pruning probability that increases with decreasing significance: if $n$ is the significance, where $n = 0$ for the most-significant element, the probability of skipping an element value having that significance is given by $1 - e^{-a_c n}$, where $a_c$ is a parameter that is used to adjust the degree of pruning.

Table 1 illustrates the performance of partial validation on the INAP example for various values of $a_s$ and $a_c$. Experimentation has shown that setting $a_c$ to ten times the chosen value of $a_s$ gives good results. For each experiment, the table gives the number of edges in the projected reachability graph constructed during validation, the percentage of the 606 statements in the INAP VFSM that were executed, and the running time of the validation run (user time obtained with the Unix "time" command on a SparcStation 20 with 192MB of memory). Note that the last experiment, with both $a_s$ and $a_c$ set to zero, is equivalent to a traditional validation run with no pruning; since this run failed to complete even after letting it run overnight, we terminated it prematurely. The results show that partial validation obtains excellent coverage, resulting in coverage of more statements in a few minutes than the traditional validation run lasting many hours.

## 8 EXAMPLES AND RESULTS

The INAP state machine has 140 VFSM states and 187 input variables (virtual inputs). We used our technique to generate a set of test sequences for a projected reachability graph with 296 423 nodes and 298 244 edges from a validation run in which 541 out of 606 statements (89.3%) of the INAP VFSM were executed. It can be easily observed that the graph is very sparse and that a lot of nodes have degree 2. A total of 83 test sequences were produced; the average length of

```
MESSAGE: COMP_LAST
STATE SEQUENCE: Scheck_type Scomponent Send Sstart Scheck_type
ADDED_TO_VIR: ITlast_component
DELETED_FROM_VIR: ITnot_last_component

MESSAGE: annCancel_op
STATE SEQUENCE: Scheck_type Soperation ScAnnCancel_op Scheck_type
ADDED_TO_VIR: Iann_saved Ibuffcancel I_sp_res_rep_not_pending
DELETED_FROM_VIR: Inoann_saved Iipcancel Isp_res_rep_pending
```

**Figure 7** Example Test Sequence

a sequence was approximately 17 messages. Figure 7 shows a portion of one of the sequences. Each entry in a sequence begins with a received message, and then indicates the sequence of VFSM states traversed by the INAP VFSM and the virtual inputs that were added to and deleted from the VIR when processing the message. The choice-vector inputs are not shown explicitly.

Although many of the test sequences represented valid scenarios of interactions of INAP and its environment, others revealed some problems with our technique. Some of the test sequences consisted of environment behavior that is impossible in the actual INAP implementation setting. These scenarios arose because the VFSM environment machines used for validation did not correctly model all aspects of the real INAP environment; further work is needed to bring the environment machines into accord with reality.

Another problem is that some of the test sequences are incomplete: they do not represent full start-to-finish call scenarios. This is due in part to the use of search-path pruning, which may in some cases cause all successor paths from a given global state to be truncated. Better results would be obtained by assuring that at least one successor of each global state is not truncated. Also, the "end points" of a scenario are not always obvious and need to be more clearly identified to our tool.

## 9 CONCLUSIONS

We have presented an automatic method for the generation of test sequences that relies on the exhaustive state-space search performed by formal validation. Although we have presented our method in the context of VFSM, it can also be applied to any notation based on extended finite state machines for which a formal validation tool exists. Despite the well-known shortcomings of validation when applied to large applications, we have shown that our method is effective even for the largest VFSM example we have encountered in practice.

We obtained a maximum coverage of around 90% of the code of INAP, which is more than adequate for real-life testing purposes; however, 100% coverage is a desirable goal. We would like to implement some more sophisticated heuristics to obtain better code coverage from partial validation. Some of the statements missed in the validation run we presented in Section 8 were

in fact executed in other runs with different random-number-generator seeds and values of the pruning parameters $a_s$ and $a_c$. Better coverage could therefore be obtained by somehow merging the results of multiple validation runs.

## ACKNOWLEDGMENTS

We would like to thank Pat Polsley and Costas Tsioras for their help in understanding and validating the VFSM implementation of INAP. The third author would also like to thank Tom Ball for many helpful discussions.

## REFERENCES

Aho, A. V., Dahbura, A. T., Lee, D., and Uyar, M. U. (1991) An optimization technique for protocol conformance test generation based on UIO sequences and rural Chinese postman tours. *IEEE Trans. on Communication*, vol. 39, no. 11, pp. 1604-15.

Chanson, S. T., and Zhu, J. (1993) A unified approach to protocol test sequence generation. *Proc. INFOCOM*, pp. 106-14.

Chow, T. S. (1978) Testing software design modeled by finite-state machines. *IEEE Trans. on Software Engineering*, vol. SE-4, no. 3, pp. 178-87.

Cormen, T. H., Leiserson, C. E., and Rivest, R. L. (1990). *Introduction to Algorithms*, The MIT Press.

Flora-Holmquist, A. R., O'Grady, J. D. and Staskauskas, M. G. (1995a) Telecommunications software design using virtual finite state machines. *Proc. Intl. Switching Symposium (ISS'95)*, Berlin, Germany.

Flora-Holmquist, A. R., and Staskauskas, M. G. (1995b) Formal validation of virtual finite state machines. *Proc. Workshop on Industrial-Strength Formal Specification Techniques (WIFT95)*, Boca Raton, FL, pp. 122-9.

Flora-Holmquist, A., and Staskauskas, M. (1996) Moving formal methods into practice: The VFSM experience. *Proc. Workshop on Formal Methods in Software Practice*, San Diego, CA, pp. 49-59.

Fujiwara, S., v. Bochmann, G., Khendek, F., Amalou, M., and Ghedamsi, A. (1991). Test selection based on finite state models. *IEEE Trans. on Software Eng.*, vol. 17, pp. 591-603.

Hennie, F. C. (1964) Fault detecting experiments for sequential circuits, *Proc. 5th Ann. Symp. Switching Circuit Theory and Logical Design*, pp. 95-110.

Holzmann, G. J. (1991) *Design and Validation of Computer Protocols*, Prentice-Hall.

European Telecommunications Standards Institute (ETSI) (1994) Intelligent Network; Intelligent Network Capability Set 1 (CS1); Core Intelligent Network Application Protocol (INAP); Part 1: Protocol Specification. Draft standard prETS 300 374-1.

International Telecommunications Union (ITU) (1993) Distributed Functional Plane for Intelligent Network CS-1. ITU-T Recommendation Q.1214.

Lee, D., Sabnani, K.K., Kristol, D.M., and Paul, S. (1996) Conformance testing of protocols specified as communicating finite state machine s - a guided random walk based approach. *IEEE Trans. on Communications*, vol. 44, no. 5, pp. 631-640, 1996.

Lee, D., and Yannakakis, M. (1996a). Principles and methods of testing finite state machines—A survey. *Proceedings of the IEEE*, August 1996.

Lee, D., and Yannakakis, M. (1996b). Optimization problems from feature testing of communication protocols. *Proc. of The 4th International Conference on Network Protocols*, Columbus, 1996.

Naito, S., and Tsunoyama, M. (1981) Fault detection for sequential machines by transitions tours. *Proc. IEEE Fault Tolerant Comput. Symp. IEEE Computer Society Press,* pp. 238-43.

Sabnani, K. K., and Dahbura, A. T. (1988) A protocol test generation procedure. *Computer Networks and ISDN Systems,* vol. 15, no. 4, pp. 285-97.

Sidhu, D. P., and Leung, T.-K. (1989) Formal methods for protocol testing: a detailed study. *IEEE Trans. Soft. Eng.*, vol. 15, no. 4, pp. 413-26.

Telelogic AB (1996) TTCN Link—A tool for test suite development. World Wide Web page, URL: `http://www.telelogic.se/products/ttcnlink.htm`.

Teradyne, Inc. (1996) TestMaster automated test generation system. World Wide Web page, URL: `http://www.teradyne.com/sst/ssthome.html`.

Wagner, F. (1992) VFSM executable specification. *Proc. IEEE International Conference on Computer System and Software Engineering*, pages 226-231, The Hague, 1992.

West, C. (1986) Protocol validation by random state exploration. *Proc. IFIP WG6.1 6th Intl. Symp. on Protocol Specification, Testing, and Verification*, North-Holland, B. Sarikaya and G. Bochmann, Ed. 1986.

Wiles, A., Ek, A., and Ellsberger, J. (1993) Experience with computer-aided test suite generation. *Proc. 6th IFIP Workshop on Protocol Test Systems*, Pau, France.

Yannakakis, M., and Lee, D. (1995) Testing finite state machines: fault detection. *J. of Computer and System Sciences*, Vol. 50, No. 2, pp. 209-227.

## 10 BIOGRAPHIES

Samuel Huang received the Ph.D. degree in mathematics from State University of New York in 1978, and the M.S. degree in computer science from University of Wisconsin in 1988. He is presently a member of the technical staff in the 5ESS Call Processing Features Development department at Lucent Technologies Inc. Naperville, IL.

David Lee received the M.A. degree in mathematics from the City University of New York in 1982 and the Ph.D. degree in computer science from Columbia University in 1985. Since 1985, he has been a member of technical staff at the Computing Science Research Center at Bell Laboratories, Lucent Technologies Inc., Murray Hill, NJ. He has been an adjunct professor at Columbia University. His current research interests are communication protocols and complexity theory.

Mark Staskauskas received the Ph.D. degree in computer sciences from the University of Texas at Austin in 1992. Since then, he has been a member of the technical staff in the Software Production Research Department at Bell Laboratories, Lucent Technologies Inc., Naperville, IL. His research interests include formal methods for concurrent programs, software testing, and software engineering.