PYTHIA: A SOFTWARE TOOL FOR TESTING DATA PORTIONS OF NETWORK PROTOCOLS

David Lee and Mihalis Yannakakis *

May 27, 1999

Abstract

As computer network systems become more complex testing is indispensable for their reliability and performance. Much has been done on testing of protocol control portions yet little is known about data portion testing. A software tool **PYTHIA** was developed at Bell Laboratories for generating test sequences automatically for protocol data portions; it minimizes the number of tests and guarantees a desirable fault coverage. We present the algorithms for the test generation, analyze the complexity of various problems involved, and report experiments on real systems, including PHS, 5ESS INAP, and ATM PNNI protocols.

1 Introduction

A complex communication system uses a set of rules, called a *protocol*, to define the interactions among the system entities. To meet the ever increasing demand from the users and to fulfill complicated tasks protocol systems are getting larger and more complex, yet they are also becoming less reliable. Consequently, testing is an indispensable part of system design and implementation, however, it has proved to be a formidable task for complex network systems.

A protocol system is implemented according to a specification, which can be a design requirement or standard. To ensure reliable communications among the implementations from a same specification, they must be tested for conformance to the specification. Typically, an implementation is tested by an external tester; the tester applies a stimulus to the implementation under test and verifies that the corresponding system response is that which is expected. 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 *developer's* test. There are high level tests such as *deliverable* and *feature* test. In this work we are focused on high level testing.

*Bell Laboratories, Lucent Technologies, Murray Hill, New Jersey

A typical protocol system has a number of system states, which are often called *control* states. With an external stimulus, which is an *input* to the system, the protocol system moves to another state via a *transition* between the two states, and provides a system response, which is an *output*. This *control portion* of the system behavior can be mathematically modeled by a *finite state machine*. [1]

Most of the formal work on conformance testing addresses the problem of testing the control portion, i.e., testing of finite state machines. 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 instance, the IEEE 802.2 Logical Link Control Protocol (LLC) [7] 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 [13, 22], the U-method based on UIO sequences [1, 32] the W-method based on characterization sets [10], and the T-method based on transition tours [30]. A general polynomial time test generation algorithm was reported in [39]. For a survey, see [19, 23, 33].

A protocol system typically has variables and parameters, and transitions between control states are associated with conditions and actions on variable values. This is often called the *data portion* of a protocol system. Finite state machines model well control portions of communication protocols. However, the data portions include variables and operations based on their values; ordinary finite state machines are not powerful enough to model in a succinct way the physical systems any more. *Extended finite state machines*, which are finite state machines extended with variables, have emerged from the design and analysis of communication protocols [14, 23]. For instance, IEEE 802.2 LLC is specified by 14 control states, a number of variables, and a set of transitions (pp. 75-117). For example, a typical transition is (p. 96):

TRANSITION
current_state SETUP
input ACK_TIMER_EXPIRED
predicate S_FLAG=1
action P_FLAG:=0; REMOTE_BUSY:=0
output CONNECT_CONFIRM
next_state NORMAL

In state SETUP and upon input ACK_TIMER_EXPIRED, if variable S_FLAG has value 1, then

[1] In the introduction we use standard terms such as finite state machine. They will be defined in Section 2.

the machine sets variables P_FLAG and REMOTE_BUSY to 0, outputs CONNECT_CONFIRM, and moves to state NORMAL.

To model this and other network protocols, such as IP protocols, other ISO standards, and complicated systems such as 5ESS, [2] we use extended finite state machines, which model well both the control and data portions of network protocols.

Our work was motivated by the efforts in test generation for Personal HandyPhone System (PHS), which is a 5ESS based ISDN wireless system, a product of AT&T/Lucent, 5ESS Intelligent Network Application Protocol (INAP), and ATM PNNI protocols. We model these systems by extended finite state machines for testing of the data portions of the protocols. We have developed efficient algorithms for test generation; we want to construct a minimal number of test sequences with a desired fault coverage. The fault coverage depends on the requirements from the domain experts and execution environment among other needs, and may vary greatly with different systems. We present test generation algorithms based on a general criterion. We then address this issue in details when we discuss experiments on real systems. It deserves a comment that we want to minimize the number of tests. If testing is conducted in software simulated environment, the number of tests usually is not a serious concern. However, in the field testing, i.e., tests are performed on real systems and devices, one has to set up the execution environment by resetting the systems etc., each test is time consuming and expensive. Under such circumstances, to minimize the number of tests is a *must*.

Based on our test generation technique, we have built a software system, called **PYTHIA**, which generates test sequences automatically for protocol data and control portions. There is a Graphical User Interface (GUI) for the conveniences of the users. One can input a protocol system through the GUI, or the GUI can pick up a protocol specification, which is written in certain machine readable formats. **PYTHIA** then generates test sequences automatically, creates a file of the generated tests, and displays them on the GUI. The GUI is similar to that of other commercial tools and there is no research content. The interested readers are referred to [37]. We shall be focused on the algorithms for test generation and on the experiments on real systems.

Basic concepts are described in Section 2. The algorithms for test generation are given in Section 3. Experimental results on PHS, INAP and ATM PNNI are presented in Section 4.

2 Basics

We first define finite state machine and extended finite state machine that model protocol control and data portions, respectively. We then describe conformance testing and reduce the problem to an optimization on graph path covering.

[2] AT&T/Lucent No. 5 Electronic Switching System

2.1 Protocol Control Portion and Finite State Machine

Protocol control portion has a finite number of control states. Upon receiving an input, it moves to another state and produces an output as a system response to the input. Such system behavior can usually be modeled by *Mealy* machines that produce outputs on their state transitions after receiving inputs:

Definition 1 A finite state machine (FSM) M is a quintuple $M = (I, O, S, \delta, \lambda)$ where I, O, and S are finite and nonempty sets of input symbols, output symbols, and states, respectively. $\delta : S \times I \rightarrow S$ is the state transition function; and $\lambda : S \times I \rightarrow O$ is the output function. When the machine is in a current state s in S and receives an input a from I it moves to the next state specified by $\delta(s, a)$ and produces an output given by $\lambda(s, a)$.

□

2.2 Protocol Data Portion and Extended Finite State Machine

The data portion of a protocol system contains system variables. [3] Furthermore, transitions are associated with conditions and actions on the variable values in addition to inputs and outputs. We model the system data portion behaviors by extend finite state machines as follows. We denote a finite set of variables by a vector: $\vec{x} = (x_1, \ldots, x_k)$. A predicate on variable values $P(\vec{x})$ returns FALSE or TRUE; a set of variable values $\vec{x}$ is valid for P if $P(\vec{x}) =$ TRUE, and we denote the set of valid variable values by $X_P = \{\vec{x} : P(\vec{x}) = \text{TRUE}\}$. Given a function $A(\vec{x})$, an action (transformation) is an assignment: $\vec{x} := A(\vec{x})$.

Definition 2 An *extended finite state machine* (EFSM) is a quintuple $M = (I, O, S, \vec{x}, T)$ where I, O, S, $\vec{x}$, and T are finite sets of input symbols, output symbols, states, variables, and transitions, respectively. Each transition t in the set T is a 6-tuple:

$$t = (s_t, q_t, a_t, o_t, P_t, A_t)$$

where s_t, q_t, a_t, and o_t are the start (current) state, end (next) state, input, and output, respectively. $P_t(\vec{x})$ is a predicate on the current variable values and $A_t(\vec{x})$ defines an action on variable values.

Initially, the machine is in an initial state $s_0 \in S$ with initial variable values: $\vec{x}_{init}$. Suppose that the machine is at state s_t with the current variable values $\vec{x}$. Upon input a_t, if $\vec{x}$ is valid for P_t, i.e., $P_t(\vec{x}) =$ TRUE, then the machine follows the transition t, outputs o_t, changes the current variable values by action $\vec{x} := A_t(\vec{x})$, and moves to state q_t.

[3] Often it also contains parameters which are in the input or output messages and may change variable values and influence system behaviors. For clarity, we study extended finite state machines with variables only and comment later on parameters.

For each state $s \in S$ and input $a \in I$, let all the transitions with start state s and input a be: $t_i = (s, q_i, a, o_i, P_i, A_i)$, $1 \leq i \leq r$. In a *deterministic* EFSM the sets of valid variable values of these r predicates are mutually disjoint, i.e., $X_{P_i} \cap X_{P_j} = \emptyset$, $1 \leq i \neq j \leq r$. Otherwise, the machine is *nondeterministic*.

□

In a deterministic EFSM there is at most one transition to follow at any moment, since at any state and upon each input, the associated transitions have disjoint valid variable values for their predicates and, consequently, current variable values are valid for at most one predicate. On the other hand, in a nondeterministic EFSM there may be more than one possible transition to follow. In this paper we only consider deterministic EFSM's. Clearly, if the variable set is empty and all predicates $P \equiv$ TRUE then an EFSM becomes an ordinary FSM.

Example 1 A simple vending machine sells coke for 10c a bottle and only takes dimes and nickels. When the accumulated credits is 10c or more the machine drops a bottle of coke and changes if any. Before the credits reach 10c, a *refund* button can be pushed to get the changes back. We model its behavior by an EFSM M with inputs $I = \{dime, nickel, refund\}$, for inserting dime, nickel, and pushing *refund* button, respectively, and with outputs $O = \{dime, nickel, coke\}$, for dropping dime, nickel, and a bottle of coke, respectively. The *transition diagram* in Figure 1.1 specifies the machine. It has control states $S = \{s_0, s_1\}$ and an integer valued variable x that keeps track of the credits. The initial state is s_0 and the initial variable value is $x = 0$. There are 5 transitions and each transition is represented by an arc from a start state to an end state. The rest of the components of a transition is associated with the arc in a format $input, predicate/output, action$ where $-$ stands for the absence of input, output, or action.

fig1.1.ps

□

2.3 Reachability Graph

Given an EFSM $M = (I, O, S, \vec{x}, T)$, each combination of a state and variable values is called a *configuration* (or *context*). Given a configuration $[s, \vec{x}]$ and any transition from state s: $t = (s, q, a, o, P, A)$, if $P(\vec{x}) =$ TRUE then upon input a we can execute transition t in machine M, update variable values by action $A(\cdot)$: $\vec{x} := A(\vec{x})$ move to state q, and output o. The configuration of the ending state and updated variable values is: $[q, A(\vec{x})]$. Naturally, transition t induces a transition $\bar{t}$ from configuration $[s, \vec{x}]$ to $[q, A(\vec{x})]$. This transition $\bar{t}$ from configuration $[s, \vec{x}]$ to $[q, A(\vec{x})]$ "inherits" the input a and output o from transition t of the EFSM M. Since the predicate P on transition t has been "checked" to be TRUE for the variable values $\vec{x}$ and the action A has been "taken" by assigning $A(\vec{x})$ to the ending configuration $[q, A(\vec{x})]$, the predicate P and the action A become redundant, and we omit both in transition $\bar{t}$. Therefore, given an EFSM, if each

variable has a finite number of values (Boolean variables for instance), then there is a finite number of configurations. Include all the induced transitions between the configurations, we obtain an equivalent (ordinary) FSM with configurations as states. Therefore, an EFSM with finite variable domains is a compact representation of an FSM. [4]

An EFSM usually has an initial state s_{init} and all the variables have an initial value $\vec{x}_{init}$; they give the *initial configuration*: $v_{init} = [s_{init}, \vec{x}_{init}]$. We are only interested in the configurations that are reachable from the initial configuration, i.e., there is a path along the transitions from the initial configuration to that configuration. Take all the reachable configurations and the transitions between them, and we obtain a *reachability graph* of the EFSM. Obviously, a reachability graph is also an FSM.

Example 1 (con't) The reachability graph of machine M in Fig. 1.1 of Example 1 is shown in Figure 1.2. It contains 6 configurations and 10 induced transitions, each of which is labelled with the original transition of M. The initial configuration is $[s_0, 0]$. □

2.4 Conformance Testing

Conformance testing checks whether an implementation conforms to its specification. Specifically, we have a specification machine, which is a design of a protocol system, and an implementation machine, which is a "black box" for which we can only observe its input/output (I/O) behavior, we want to test whether the implementation *conforms* to the specification. This is called *conformance testing* or *fault detection*.

Let the specification and implementation EFSM be A and B, and let their corresponding reachability graph be $\bar{A}$ and $\bar{B}$. Since A (B) and $\bar{A}$ ($\bar{B}$) are equivalent, we can test conformance of the implementation FSM $\bar{B}$ to the specification FSM $\bar{A}$ instead, using the known methods for FSM testing. For instance, we may consider a *checking sequence* [10, 13, 19, 23, 29, 35, 39], which guarantees that the implementation machine $\bar{B}$ is structurally isomorphic to the specification machine $\bar{A}$. However, even for medium size machines it is too long to be practical [39] while for EFSM's hundreds of thousands of configurations are typical and it is in general impossible to construct a checking sequence. From our experiences in real protocol system testing, structural isomorphism of the implementation EFSM and the specification EFSM is not only impossible to test but also an "overkill"; the desired fault coverage and hence the test sequences are different than that for FSM's.

Definition 3 Let A be a specification EFSM with an initial state s_{init} and variable values $\vec{x}_{init}$. The corresponding reachability graph $\bar{A}$ has an initial configuration $[s_{init}, \vec{x}_{init}]$. A *test sequence* (or a *scenario*) is a path in $\bar{A}$ from the initial configuration to a configuration with the initial state $[s_{init}, \vec{x}]$ where $\vec{x}$ may be different than $\vec{x}_{init}$.

[4] From Definition 2 the equivalent FSM of an EFSM may not be completely specified as defined in Definition 1.

□

We are only interested in testing systems with deterministic behaviors, i.e., at a state of the EFSM and upon an input there is at most one transition from that state, which is executable. Therefore, given an input sequence, there is at most one path in the reachability graph from the initial configuration that is labelled with the given input sequence. We can use the term "input sequence" and "path from the initial configuration in the reachability graph" interchangeably when we discuss test sequences.

We want to construct a set of test sequences of a desirable *fault coverage* which ensures that the implementation machine under test *conforms* to the specification. The fault coverage is essential, and is often defined differently from different models and/or practical needs. From our experiences with protocol systems a practical criterion of fault coverage is that each transition in the specification EFSM has to be executed at least once. For each transition in the EFSM, we can designate a distinct *color* and it is also carried over to the induced transitions in the reachability graph. We want to find tests such that all the colors are covered, i.e., the corresponding transitions are executed. Note that a same color may have multiple appearances in the reachability graph. For some protocol systems, we want to test some variable values; typically, they are boundary values. A certain combination of variable values corresponds to configurations with that variable values (but may have different control states). We can also designate a distinct color for those configurations, and find tests that cover all these colors. For those configurations (transitions) without designated colors, i.e., we do not care whether they are tested, we assign a unique color: *white*. In summary, given a specification EFSM A with a reachability graph $\bar{A}$, each configuration and transition of $\bar{A}$ has a distinct color or *white*; these colors are from practical needs of fault coverage. We call it a *color assignment* of EFSM A, and denote the set of all colors by C. A color c is *covered* by a test sequence if the corresponding path in the reachability graph contains a configuration or transition with color c. We want to construct a set of test sequences that covers all the colors in C at least once:

Definition 4 A *complete test set* for conformance testing of an EFSM with a color assignment of color set C is a set of test sequences such that each color in C is covered by at least one of test.

□

Given the succinct representation of EFSM's, one might imagine that it is an *easy* problem. As a matter of fact, even an apparently easier problem, the reachability problem, is hard where we want to determine if a control state is reachable from the initial state. Specifically, the Turing machine halting problem can be reduced to the reachability problem of EFSM, which is, therefore, undecidable if the variable domains are infinite and PSPACE-complete otherwise [15].

From Definition 3 and 4, for test generation we only need to examine the specification EFSM with an initial state s_{init} and variable values $\vec{x}_{init}$. It has a reachability graph G with a *source node*, which is the initial configuration $v_0 = [s_{init}, \vec{x}_{init}]$. For clarity, we assume that each test sequence

is from the initial node to a *sink* node v_∞, which is a configuration with the initial control state by Definition 3. We use term *edges* for the induced transitions in G. As for the color assignment, we consider a more general case; each node and edge contains a subset of colors from the color set C. We want to find a set of paths from source to sink node such that all the colors in C are covered. We thus reduce the testing problem to a graph path covering problem.

3 Test Generation Algorithms

Formally, we have a directed graph $G =< V, E >$ with $n = |V|$ nodes, $m = |E|$ edges, a *source* node v_0 of in-degree 0, and a *sink* node v_∞ of out-degree 0. All edges are reachable from the source node and the sink node is reachable from all edges. There is a set C of $k = |C|$ distinct *colors*. Each node and edge 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 complete test set that cover all the colors; they are not necessarily the conventional covering paths that cover all the edges. The test (path) length makes little difference and we are interested in minimizing the number of paths. Consequently, we can shrink each strongly connected component (SCC) [2] 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 directed acyclic graph (DAG). From now on, unless otherwise stated, we assume that the graph $G =< V, E >$ is a DAG.

Example 1 (con't) For the EFSM in Example 1, if we want to test each of the transitions and variable values, we need 10 colors: $\{0, 1, \ldots, 9\}$; color i is for transition t_i, $i = 0, 1, \ldots, 5$, and color $6, 7, 8, 9$ are for variable values $0, 5, 10, 15$, respectively. The corresponding color assignment is in Fig. 1.2. The following 2 tests cover all the colors but not all the edges of G: $T_1 = (v_0, v_1, v_2, v_4, v_5, v_0)$ and $T_2 = (v_0, v_5, v_0)$. □

3.1 Minimal Complete Test Set

We need a complete test set - a set of paths from the source node to the sink node that cover all the colors. On the other hand, we want to minimize the number of tests:

Problem 1 Find a complete test set of minimum cardinality.

□

The two tests in the above example provide a minimal complete test set. However, in general, it is a hard problem:

Proposition 1 Problem 1 of finding a complete test set of minimum cardinality is NP-hard.

Proof: . The *Set Cover* problem [18, 12], which is known to be NP-hard, can be reduced to this problem. Recall the Set Cover problem: given a family F of sets over a (finite) set U of elements and an integer κ, is there a covering subfamily of no more than κ members, i.e. are there κ sets in F whose union is U? In the reduction we let U be the set of colors, We construct a graph with two nodes: the source and sink, and parallel edges from source to sink such that each set in F is associated with an edge. A solution of Problem 1 provides an answer to the Set Cover problem. This construction uses parallel edges and has multiple colors associated with an edge, but it is easy to modify it, if we wish, so that the graph is simple (i.e. has no parallel edges) and each edge has only one associated color: simply replace the edges by paths of appropriate length. □

We can solve Problem 1 in time $O(m2^{O(k)})$, i.e. exponential in the number of colors and linear in the size of the graph, by a bottom-up, essentially brute-force approach. Since G is a DAG, we can topologically sort the nodes (see eg. [9]. Compute a family F_u of color sets for each node u in the reverse topological order as follows. The sink node contains a singleton set of all its colors. When processing a node u, we examine all its outgoing edges (u, v); note that node v has been processed and has already a family F_v of color sets. For each set of F_v, we add the colors of edge (u, v) and that of node u, and obtain a set of colors. We collect all the color sets from all the outgoing edges from u and obtain a family of color sets for u. We can obviously remove duplicates, so $|F_u| \leq 2^k$.

Besides duplicates, we can also remove a color set which is included in another one, i.e., if there are two colors sets C_i and C_j at a node u with $C_i \subseteq C_j$ then we can discard C_i. Therefore, at each node, we only have to a keep a record of color sets, none of which is a subset of the other. Given a set C of cardinality $k = |C|$, denote by $\mu(k)$ the maximum number of subsets of C such that none is a subset of another. It is well known (see eg. [26]) that the maximum is achieved by the family of all subsets of cardinality $\lfloor k/2 \rfloor$ and $\mu(k) = \begin{pmatrix} k \\ \lfloor k/2 \rfloor \end{pmatrix}$.

Obviously, the family F_u of color sets of u has the following properties: (1) For each color set, there is a path from u to the sink whose colors are the same as the color set; and (2) The colors on any path from u to the sink are contained in one of the color sets in u. This can be easily proved by an induction on the ordering of processing. For an efficient path construction from the color sets, we associate each color set c of u with the edge (u, v) and the color set in F_v, from which we have obtained the color set c, so that we can trace the path corresponding to each color set of u.

At the source node s, we find a minimal number of color sets in F_s that covers all the colors. The corresponding paths can be constructed from the edges associated with the color sets. This gives a complete test set of minimum cardinality.

We discuss the time complexity. When we process a node u, for each outgoing edge (u, v), we merge the color sets associated with node u, and edge (u, v), with every set of the family F_v for node v, which has been processed. Although the color sets that are contained in other sets are not needed and can be discarded, it takes some effort to find and eliminate them. We can choose to apply

the algorithm either with the elimination of subsets at the nodes, or without subset elimination (i.e. only duplicate elimination). Let f, f' respectively be an upper bound on the number of color sets at each node in these two cases; thus, $f \leq f'$, $f \leq \mu(k)$ and $f' \leq 2^k$ where k is the number of colors. We can represent a color set by a list of its elements or by its characteristic vector. We can represent a family of color sets (eg. F_u) by a trie. Checking whether a color set is in a family F and adding it if it is not takes time $O(k)$. Thus, if we do not eliminate subsets, then the operation on each edge takes time proportional to $f'k$ and the total cost is $O(f'km)$, where m is the number of edges of the graph. If we eliminate subsets, then the straightforward implementation of each edge operation takes time proportional to f^2k and the total cost is $O(f^2km)$. The quadratic factor is because for each set that is added to a family we have to compare it for inclusion with every set of the family. Unfortunately there is no good way known to find the maximal sets of a given family (i.e., those that are not contained in another set) in time substantially better than the straightforward quadratic time; a small improvement (by a factor of $\sqrt{logf}$) is given in Subset elimination is advantageous only if $f^2 < f'$. Note that for the worst case upper bounds of f and f' we have $\mu^2(k) >> 2^k$. In any case, without subset elimination we can compute the family F_s for the root s in worst case time complexity $O(2^kkm)$ over the whole graph.

At the source node s, we select a minimal number of color sets from the family F_s that cover all the k colors. This can be done in time $O(2^2f'k) = 2^{O(k)}$. Consider a graph that has one node for every subset of colors, and has an edge from node P to node Q if $Q = P \cup R$ for some $R \in F_s$. The graph has 2^k nodes and out-degree f', thus it has $2^kf'$ edges. A minimum cover corresponds to a shortest path in the graph from node $\emptyset$ to the node C of all the colors. The standard breadth-first search algorithm will find the shortest path in linear time in the size of the graph. Once the minimum cover is found, the corresponding paths of the DAG can be traced from the edges associated with the color sets with a cost proportional to the total lengths of the paths. In summary:

Proposition 2 We can solve Problem 1 of finding minimal complete test set in time $m2^{O(k)}$, and space $O(m)$ where m is the number of the edges of the graph and k is the total number of colors to be covered.

□

3.2 Maximal Color Paths

The cost of the above approach is exponential in the number of colors k and thus it is feasible only for small values of k. For moderate and large number of colors we need to restrict ourselves to approximation algorithms. Our problem is a Set Cover problem where the sets are not given explicitly, but rather are defined implicitly by the paths of the graph. The standard approximation algorithm for Set Cover [17, 25] is the following greedy method: pick a set of maximum cardinality

from the given family, then pick a set that covers the maximum number of uncovered elements, and continue similarly picking sets until all the elements are covered. This algorithm corresponds to the following greedy method for our problem. We first find a path (test) that covers a maximum number of colors and delete the covered colors from C. We then repeat the same process until all the colors have been covered. Thus, we have the following problem:

Problem 2 Find a test that covers the maximum number of uncovered colors.

□

This problem is also NP-hard. Our reduction is from the maximum independent set problem in regular graphs. In this problem we are given a regular undirected graph H (i.e. all the nodes have the same degree d) and a positive integer l, and we wish to determine if there is a independent set I of at least l nodes (i.e., no two nodes of I are adjacent). Construct a DAG G that consists of a source node s, sink node t, and l levels of n nodes each, where n is the number of nodes of H. Thus, every node u of G (except s and t) in each level corresponds to a node $h(u)$ of H. There are edges from s to all the nodes of the first level, edges from all the nodes of each level to all the nodes of the next level, and finally edges from the nodes of the last level l to node t. We let the set of colors be the set of edges of the given graph H. The set of colors associated with all edges of G and with nodes s and t is $\emptyset$, and the set of colors associated with any other node u of G is the set of edges of H incident to the corresponding node $h(u)$ of H.

We claim that H has an independent set with l nodes if and only if G has a path that covers dl colors. From the construction it is clear that there is a 1-to-1 correspondence between $s-t$ paths of G and sequences of l (not necessarily distinct) nodes of H. The set of colors covered by a path in G is the set of edges of H incident to the nodes of the corresponding sequence. Since H is a regular graph of degree d, such a set of edges (colors) can have at most cardinality dl, and furthermore, it has cardinality exactly dl iff the l nodes of H do not share any edge (implying in particular that they are distinct), i.e., iff the l nodes form an independent set. In summary,

Proposition 3 The problem of finding a test that covers the maximum number of uncovered colors is NP-hard.

□

Thus, in view of the NP-hardness of Problem 2 we have to content ourselves with approximation algorithms. We now describe some heuristic methods.

3.2.1 Longest Path

For a node v denote the set of uncovered colors in v by $c(v)$. Similarly, $c(u,v)$ denotes the set of uncovered colors in edge (u,v). We assign a weight to an edge (node), which is the cardinality of the uncovered colors in that edge (node), i.e., $|c(v)$ $(|c(u,v)|)$, and we obtain a weighted graph. We

find a longest path from the source to sink in this weighted graph; it is possible since it is a DAG. This may not provide a maximal color test due to the multiple appearances of colors on a path. However, if there are no multiple appearances of colors on the path, then it is indeed a maximal color test.

There are known efficient ways of finding a longest path on a DAG. We can first topologically sort the nodes and then compute the longest paths from each node to the sink in the reverse topological order.

Algorithm 1

input. a weighted DAG G with source v_0 and sink v_∞.

output. a longest path from source to sink.

1. topologically sort nodes in G: $v_0, v_1, \ldots, v_{n-1} = v_\infty$;
2. $l(v_{n-1}) = 0$, $p(v_{n-1}) = \text{NULL}$;
3. **for** $i = n-2, \ldots, 0$
4. $\quad l(v_i) = 0$; $p(v_i) = \text{NULL}$;
5. $\quad$ let outgoing edges from v_i be: $w_1, \ldots, w_r$;
6. $\quad$ **for** $j = 1, \ldots, r$
7. $\quad\quad$ **if** $|c(v_i)| + |c(v_i, w_j)| + l(w_j) > l(v_i)$
8. $\quad\quad\quad l(v_i) = |c(v_i)| + |c(v_i, w_j)| + l(w_j)$, $p(v_i) = w_j$;
9. $lpath = v_0$, $v = v_0$;
10. **while** $p(v) \neq \text{NULL}$
11. $\quad v = p(v)$;
12. $\quad lpath = lpath \circ v$;
13. **return** $lpath$;

For efficient implementation, each node u records the length $l(u)$ of the longest path from that node to sink and also the next node $p(u)$ from u on the path. We construct a longest path from the nodes in a reversed topological order. Line 4-8 find a longest path from a node v_i to sink. For each outgoing edge (v_i, w_j) the longest path from w_j to sink has been found since w_j has larger topological order than v_i. Obviously, the chosen node $p(v_i)$ is the next node on the longest path from v_i to sink. After processing node v_0 the longest path from the source v_0 has been found, and it is constructed in Line 9-12 by following the next nodes on the path, recorded in $p(\cdot)$.

The topological sort takes time $O(m)$. During the path finding process, each edge is examined at most once. Hence,

Proposition 4 Algorithm 1 finds an approximate maximum color path in time $O(m)$ where m is the number of edges of the graph.

□

How does this heuristic method compare with the optimal solution? In the worst case it can be really bad.

Example 2 A graph G consists of $k+1$ separate paths P_i, $i = 0, 1, \ldots, k$, from source to sink. Path P_0 has k edges of k distinct colors and path P_i has $k+1$ edges, all of the same color i, $i = 1, 2, \ldots, k$. Path P_o provides a minimal complete test set with only 1 test. However, the longest path heuristic will pick one of the paths with $k+1$ edges. Iterating the process, the method will require k tests corresponding to the k paths P_i, $i = 1, 2, \ldots, k$.

□

3.2.2 Greedy Heuristics

We discuss greedy heuristic procedures for finding maximal color paths. The idea is similar to the exact procedure described in Section 3.1, which takes time $O(2^k km)$. The heuristic procedures take linear time and work well in practice, see Section 4.

We again topologically sort the nodes and compute a desired path from each node to the sink in a reverse topological order as follows. Instead of keeping the color sets of all the paths from a node to the sink,

Algorithm 2

input. a colored DAG G with source v_0 and sink v_∞.

output. an approximate maximal color path from source to sink.

1. topologically sort nodes in G: $v_0, v_1, \ldots, v_{n-1} = v_\infty$;
2. $\kappa(v_i) = \emptyset$, $p(v_{n-1}) = \text{NULL}$;
3. **for** $i = n-2, \ldots, 0$
4. $\quad \kappa(v_i) = \emptyset$, $p(v_i) = \text{NULL}$;
5. $\quad$ let outgoing edges from v_i be: $w_1, \ldots, w_r$;
6. $\quad$ **for** $j = 1, \ldots, r$
7. $\quad\quad$ **if** $|c(v_i) \cup c(v_i, w_j) \cup \kappa(w_j)| > |\kappa(v_i)|$
8. $\quad\quad\quad \kappa(v_i) = c(v_i) \cup c(v_i, w_j) \cup \kappa(w_j)$, $p(v_i) = w_j$;
9. $lpath = v_0$, $v = v_0$;
10. **while** $p(v) \neq \text{NULL}$
11. $\quad v = p(v)$;
12. $\quad lpath = lpath \circ v$;
13. **return** $lpath$;

The procedure is similar to Algorithm 1 except that at each node v_i we store all the colors of the nodes and edges on an approximate maximal color path from v_i, denoted by $\kappa(v_i)$. Specifically, when we process a node v_i and consider all the outgoing edges (v_i, w_j), we take the union of the colors of node v_i, edge (v_i, w_j), and $\kappa(w_j)$. We compare the resulting color sets from all the outgoing edges from v_i and keep one with the largest cardinality. Likewise, it may not provide a maximum color coverage test; when we choose the outgoing edge from v_i, we do not incorporate information of the colors from the source to v_i. Since we take unions of and compare color sets of no more than k colors, the time and space complexity of this approach is $O(km)$.

Proposition 5 Algorithm 2 finds an approximate maximum color path in time $O(km)$ where k is the number of colors and m is the number of edges of the graph.

□

Although the second method seems to be better in many cases, its worst case ratio to the optimal solution is also $\Omega(k)$.

Example 3 Consider the DAG G shown in Figure 1.2. The colors 1,..., 6 associated with the edges are shown next to them, and the nodes have no colors. The optimal path $v_7, \ldots, v_1$ covers all the colors.

Consider the application of the greedy heuristic to this graph. When we process node v_2 we have a choice whether to keep color set 1 or 2. We have not specified how to break ties. Suppose that ties are broken in favor of higher numbered colors. Then the heuristic will pick 2 for v_2. At node v_3, there is a choice between 2 and 3, so we will pick set 3. Continuing in this manner, we see that the heuristic will find a set of cardinality 1 instead of 6. The example clearly generalizes to arbitrary number of colors k.

□

It is easy to improve on the greedy heuristic. For example if we keep at each node two sets instead of only one, then the heuristic will find the optimal path in Example 3, and in general it will be guaranteed to find a path of at least two colors if there is one. However, it is easy to construct another example for this 2-set greedy variant in which it has ratio $k/2$ to the optimal, i.e., it finds only a path of two colors while the optimal path covers all the colors. Another variant of the greedy algorithm does somewhat better.

3.2.3 A Transitive Greedy Heuristic

This is similar to the greedy heuristic, except that when we process a node u, we do not consider only its immediate successors but all its descendants.

Algorithm 3

input. a colored DAG G with source v_0 and sink v_∞.

output. an approximate maximal color path from source to sink.

1. topologically sort nodes in G: $v_0, v_1, \ldots, v_{n-1} = v_\infty$;
2. $\kappa(v_i) = \emptyset$, $p(v_{n-1}) =$ NULL;
3. **for** $i = n-2, \ldots, 0$
4. $\quad \kappa(v_i) = \emptyset$, $p(v_i) =$ NULL;
5. $\quad$ let all descendant nodes of v_i be: $w_1, \ldots, w_r$;
6. $\quad$ **for** $j = 1, \ldots, r$
7. $\quad\quad$ **if** $|c(v_i) \cup c(v_i, w_j) \cup \kappa(w_j)| > |\kappa(v_i)|$
8. $\quad\quad\quad \kappa(v_i) = c(v_i) \cup c(v_i, w_j) \cup \kappa(w_j)$, $p(v_i) = w_j$;
9. $lpath = v_0$, $v = v_0$;
10. **while** $p(v) \neq$ NULL
11. $\quad v = p(v)$;
12. $\quad lpath = lpath \circ v$;
13. **return** $lpath$;

When we process a node, we have to examine all its descendants instead of its children. Therefore, the time complexity of this algorithm is $O(knm)$.

The worst case ratio of this heuristic to the optimal is somewhat better; it is $\Theta(\sqrt{c})$, where c is the maximum number of colors covered by a path. To prove the upper bound, consider an optimal $s-t$ path p that covers c colors. For each node u, let $F(u)$ denote the set of colors assigned to u by the algorithm. Traversing the path p bottom-up from the sink t to the source s, let $usub0 = t$, let $u_1, u_2, \ldots, u_r$ be the nodes at which the cardinality of F strictly increases, and if $u_r \neq t$ let $t = u_{r+1}$; i.e. for $j = 1, 2, \ldots, r$, $|F(u_j)| > |F(u_{j-1})|$ and $|F(u)| = |F(u_{j-1}|$ for all nodes u of p between u_{j-1} and u_j. Let C_j be the set of colors that appear on the edges and nodes of p between u_{j-1} and u_j, where node u_{j-1} is excluded and u_j is included. Observe that, if the segment of p between u_{j-1} and u_j is nontrivial, i.e., contains some intermediate nodes and edges, then all their colors must be contained in $F(u_{j-1})$, because otherwise one of these intermediate nodes would be assigned a larger color set. Thus, all colors of $C_j - F(u_{j-1})$ appear on the node u_j and the edge from u_j to its child in p. From the computation of $F(u_j)$ we have, $|F(u_j) \geq |F(u_{j-1}) \cup (C_j - F(u_{j-1}))| \geq |C_j|$. Note that the number c of colors of the path p is equal to $|F(u_o) \cup_j C_j| \leq \sum_j |F(u_j)|$. Let $f = |F(t)|$ be the number of colors obtained by the heuristic. Since the sets $F(u_j)$ have distinct cardinalities, except possibly for the last two, we have $c \leq \sum_{j=1}^{f+1} j = (f+1)(f+2)/2$, and thus $f = \Omega(\sqrt{c})$. It is not hard to construct an example as in Figure 2 showing that the analysis is tight, i.e., an example where $f = O(\sqrt{c})$.

Proposition 6 Algorithm 3 finds an approximate maximum color path in time $O(kmn)$ where k is the number of colors, n is the number of nodes, and m is the number of edges of the graph. The number of colors on the constructed path is $\Omega(\sqrt{c})$ where c is the number of colors on a maximum color path.

□

3.3 Complexity

We have shown that Problem 2 of finding a maximum color path is NP-hard. Therefore, we propose heuristic procedures. They take polynomial time and space. However, they provide approximate solutions; the ratio of the colors on a maximum color path over the colors on a constructed path by the approximation algorithms varies from c to $\sqrt{c}$ where c is the number of colors on a maximum color path. Can we do better? We address several related complexity issues now.

It is worth noting that the greedy heuristic achieves a constant factor (no more than 2) approximation in the DAGs of the NP-hardness reduction; we omit the proof. (The longest path heuristic does not have this property; it does not provide any guarantees.) This is particularly interesting in view of the fact that even for this class of DAGs there is a limit on the approximation factor that can be achieved in polynomial time. This follows from the fact that the maximum independent set

problem on regular graphs of bounded degree d is MAX SNP-hard, and hence there is a constant $r > 1$ such that it is NP-hard to distinguish for a given graph H and integer l between the following two cases: (1) the case where there is an independent set of size at least l and (2) when the maximum independent set has size less than l/r [6, 31]. Consider the DAG G that is constructed in the reduction from H and l. In case (1), the optimal path covers dl colors. In case (2) it covers less than $dl/r + (d-1)l(r-1)/r$. Thus, if we can approximate the maximum number of colors of a path in G within a factor $1-(r-1)/rd$ then we can distinguish between case (1) and (2) in H.

It follows that there is an $\epsilon > 0$ (eg, $\epsilon = (r-1)/rd$) such that Problem 2 cannot be approximated in polynomial time with ratio $1-\epsilon$ unless P=NP. In other words, Problem 3 does not have a polynomial time approximation scheme. We do not know whether Problem 3 can actually be approximated within some constant factor for all instances.

We now come back to the original minimum complete test set Problem 1. Suppose that we successfully find a maximum color test repeatedly until we obtain a complete test set in N steps and that the minimum complete test set contains N^* tests. How far is N from N^*? Is there a better algorithm? It follows from results on the Set Cover problem that $N = \Theta(N^* \log\ k)$ [25, 17]. That is, on the one hand, for any instance, if we can find repeatedly maximum color tests, then the complete test set will contain at most $N^* \log\ k$ tests; moreover, an approximation within factor r for Problem 1 will yield a test set of size at most $N^* r \log\ k$. Conversely, there are instances in which even if we could find repeatedly paths that cover the maximum number of colors, the resulting test set contains $N^* \log_e k$ test (where $\log_e$ denotes the natural logarithm).

Moreover, the negative results on the approximation of the Set Cover problem [27] imply that we cannot do better than a logarithmic factor in polynomial time. That is, for any polynomial time algorithm which constructs a complete test set of cardinality N, there are cases such that $N = \Omega(N^* \log\ k)$ (unless of course P=NP).

3.4 Paths with a Constant Bound on the Number of Colors Covered

In spite of the negative results in the worst case, the longest path and greedy heuristic procedures were applied to real systems (see Section 4) and proved to be surprisingly efficient; a few tests cover a large number of colors and, afterwards, each test covers a *very small* number of colors. A typical situation is that the first 20% tests cover more than 70% of the colors. Afterwards, 80% of the tests cover the remaining 30% of the colors, and each test covers 1 to 3 colors. Consequently, the costly part of the test generation is the second part. Under these circumstances, exact procedures for either maximal color paths or minimal complete test sets are needed to reduce the number of tests as much as possible. The question is, can we obtain more efficient algorithms if we know that there is a bound on the maximum number of colors on any path that is a small constant $c << k$. We consider the following problems.

Problem 3 Suppose that a maximum color test covers no more than $c << k$ colors where c is a small constant. (1) Find a minimum complete test set; and (2) Find a maximum color test.

□

Since a maximum color test covers at most c colors, we can use the exact brute-force method as described in Section 3.1. In this case, each color set at a node contains at most c colors. (If a set contains c colors then we have obtained a solution for Problem 3(2).) Therefore, the number f of color sets recorded at each node is bounded by:

$$\mu(k, c) = \binom{k}{c} = \frac{k!}{c!(k-c)!} \approx k^c \ .$$

It takes time $O(k^{2c}m)$ and space $O(k^{2c}n)$ to construct the color sets for all nodes and we can examine the source node and find a maximal color test. Therefore, Problem 2 can be solved in time and space polynomial in the number of colors k and the size of the graph. Although the time is polynomial, c appears in the exponent with base k, and so for moderate values of k (say 100) and small values of c (eg. 5), the time is large. There is a better method to find the maximum color set. We will see later that Problem 2 can be solved more efficiently, essentially in linear time in the size of the graph for constant c.

First, let us discuss Problem 1. Once we have computed the family of color sets at the source node, we need to solve the Set Cover problem to find a subset of minimum cardinality that covers all the k colors. The complexity varies with the constant c.

For $c = 1$, the problem is trivial: since a color set (path) contains at most one color, we can simply take k distinct color sets, which provides a minimum complete test set. On the other hand, at each node we can use a bit map to record the color sets and it takes time $O(k)$ to process each outgoing edge from a node. Therefore, the total time and space complexity is $O(km)$.

For $c = 2$, Problem 1 can be solved in polynomial time as follows. Each color set contains 1 or 2 colors. None of the singleton sets is contained in another set; otherwise, they would have been removed during the process. We take all of them into the test sets. Suppose that there are $\bar{k}$ remaining colors to be covered by the two-color sets. We want to select a minimal number of them to cover these $\bar{k}$ colors. We construct an undirected graph as follows. Each of the $\bar{k}$ colors corresponds to a node and each of the two-color sets corresponds to an edge that joins the nodes corresponding to its colors. The problem then is to find a minimal set of edges that cover all the nodes. It is the well-known edge cover problem, which can be solved in polynomial time [12] by graph matching [20].

For $c \geq 3$, Problem 1 is NP-hard; this follows from the NP-hardness of the Set Cover problem even if all the sets have at most 3 elements [12].

In the remainder of this section we will describe an efficient algorithm for Problem 2, i.e. determining whether the graph contains a path that covers c colors, and finding such a path. We use techniques similar to those of [29] and [3] for finding simple paths in graphs.

If all we want to do is to find a path that covers c colors (rather than all paths), then in the bottom-up computation we do not need to keep all the color sets but only a sufficient number of them. That is, at each node u, instead of the complete family F_u of color sets of the paths starting at u, we need keep only a subfamily L_u such that if the DAG contains a path through u that covers c colors, then there is such a path whose suffix from u to the sink t uses only colors from some member of L_u.

Let us look first at some simple cases of small c.

Case $c = 2$. If F_u contains a set with two elements, then we are done; this set suffices. If all sets are singletons, then it suffices to keep in L_u only two of them (in case F_u has more than two). For, suppose that there is a path through u that covers two colors, say a path $psub1$ from s to u that covers color a and a path p_2 from u to t that covers color b. At least one of the sets in L_u is not a, and thus we can combine the corresponding $u - t$ path with the $s - t$ path p_1 to obtain a path through u that covers 2 colors.

Case $c = 3$. If F_u contains a set with 3 colors then we are done. Otherwise, we claim that we need to keep at most only 3 singleton sets and 3 pairs, thus 6 sets in all. If F_u has more than 3 singletons, then just keep any 3 of them. As far as pairs are concerned, we delete a pair $\{a, b\}$, unless there is a color $c \neq a, b$ that is contained in all the remaining pairs, i.e., they are all of the form $\{c, x\}$. Suppose that we cannot delete the pair $\{a, b\}$ and there are 3 or more other pairs besides $\{a, b\}$. Then one of them is $\{c, x\}$ with $x \neq a, b$. In that case, we can delete all the pairs except for $\{a, b\}$ and $\{c, x\}$. Therefore, if no pair can be deleted, then there can be at most three pairs.

To see why the above reduction works, consider a path through u that covers 3 colors, and let p_1 and p_2 be the prefix and suffix before and after u. If $psub1$ covers two colors a, b, then L_u contains a singleton other than a and b. Suppose p_1 covers one color c, and p_2 colors a and b. If the pair $\{a, b\}$ was deleted, then still a pair remains that does not contain c, and hence its corresponding $u - t$ path can be combined with path p_1.

Case: general c. For general values of c, we need to keep at most c singleton sets, $\binom{c}{2}$ pairs, $\binom{c}{3}$ triples, ..., c sets of size $c - 1$, thus a total of at most $2^c - 2$ sets. To see this consider the sets of F_u of size r, $r = 1, 2, \ldots, c - 1$. We can delete such a set unless there is a disjoint set of $c - r$ colors that intersects all the remaining sets of F_u of size r. Thus, if we cannot delete any more sets of size r, then we have a family of r-sets $S_1, S_2, \ldots, S_q$ and there are corresponding $(c - r)$-sets $T_1, \ldots, T_q$, such that $E_i \cap T_i = \emptyset$ for all $i = 1, 2, dots, q$, and $E_i \cap T_j \neq \emptyset$ for all $i \neq j$. The number q of such sets can be at most $\binom{c}{r}$; see eg. [26] Ex. 13.32.

To apply the above reduction process we need to determine at each node which sets are unnecessary and delete them. For small values of c we can do this easily (as we saw for $c = 2$ and $c = 3$), and thus obtain an algorithm that runs in linear time in the sze of the input (the DAG and the given color sets for the nodes and edges). For general values of c however this is not so easy. Unfortunately, the proof of the fact in [26] is nonconstructive.

We describe now an algorithm that gets around this problem. We use the color coding method of [5]. Suppose the given colored DAG contains a path p that covers c colors. Consider a random

mapping h from the set of k colors to the set $1, ...c$. For any set S of c distinct colors (in particular the set of colors covered by p), the restriction of h to S is one-to-one with probability $c!/c^c = \Theta(\sqrt{c}e^{-c})$. Applying a mapping h yields a new recoloring of the DAG with c colors. We can use the straightforward algorithm on the c-colored DAG to search in time $O(2^c m)$ for a path that covers all c colors; such a path will be found if the mapping h is one-to-one for some set S of c original colors that is covered by a path in the DAG, i.e., with probability at least $c!/c^c$. Therefore, if we apply the above procedure for a sequence of random mappings h, we will find a path with c colors with high probability after $O(e^c)$ trials, i.e. in time $2^{O(c)}m$.

The above algorithm is probabilistic, but it can be derandomized. A $c - perfect$ family H of hash functions from the set of k colors to the set $\{1, 2, \ldots, c\}$ is a family that has the property that for any set S of c colors there exists a function $h \in H$ such that h is one-to-one on S. As shown in [3] (using [34]), one can construct a $c - perfect$ family H that contains $2^{O(c)} \log k$ functions, and each function can be evaluated in constant time on each element. Using the functions of such a c-perfect family in conjunction with the straightforward algorithm on the recolored DAG yields a deterministic algorithm that finds a path of the DAG that covers c colors (if there is such a path) in time $2^{O(c)}m \log k$. That is, Problem 2 can be solved in randomized $2^{O(c)}m$ time or deterministic $2^{O(c)}m \log k$ time.

In summary,

Proposition 7 Suppose that a maximum color test covers no more than $c << k$ colors where c is a small constant and k is the total number of colors. For $c = 1, 2$, Problem 3 can be solved in polynomial time. For $c \geq 3$, Problem 3(1) is NP-hard, and Problem 3(2) can be solved in randomized $2^{O(c)}m$ time or deterministic $2^{O(c)}m \log k$ time where m is the number of edges of the graph.

□

4 Experiments

The software tool **PYTHIA** has been used for the feature testing of Personal HandyPhone System (PHS), 5ESS Intelligent Network Application Protocol (INAP), and ATM PNNI protocol. We report experimental results.

4.1 PHS

Personal HandyPhone System (PHS), which is a 5ESS based ISDN wireless system, a product of AT&T/Lucent.

We model PHS by three EFSM's with 17 variables. The reachability graphs are DAG's. We test each machine separately using the longest path and the greedy heuristic procedures of Algorithm

1 and 2. The results are similar and we report here those from the greedy heuristic. In the table, states and transitions refer to the number of control states and transitions in the EFSM, and edges and nodes concern the corresponding reachability graphs. Each test starts from the initial state with all the variable values zero and ends back to the initial state. From the product requirements we only need to test all the transitions, and, therefore, each transition has a distinct color. The number of tests in a complete test set is also listed in the table.

machine	states	transitions	nodes	edges	tests
M_1	5	33	5	33	21
M_2	14	132	196	893	58
M_3	12	106	473	2,056	31

Table 1. Complete Test Sets for PHS Machines

For Machine 1, the first test covers 9 transitions, the second test covers 5 transitions, and the remaining 19 tests cover one additional transition each. For Machine 2, the first 7 tests cover 78 transitions, and the remaining 54 transitions are covered by 51 tests where each test covers no more than 3 transitions. For Machine 3, the first 5 tests cover 73 transitions, and the remaining 33 transitions are covered by 26 tests where each test covers no more than 3 transitions.

In parallel with our efforts, a test set has also been generated manually with a complete coverage and contains approximately 200 test sequences. The total number of our tests is 110. As a matter of fact, after first few tests, the test sequences from our algorithm are minimal and complete test sets for the remaining untested transitions.

4.2 INAP

5ESS Intelligent Network Application Protocol (INAP) is a complex protocol, it provides credit/calling card calls, 800 calls, 911 calls, call forwarding/screening, and many other features.

We model INAP protocol by an EFSM, which interacts with two other *environment machines*. The coverage required is not all the transitions but the code in the specification in VFSM notation (Virtual Finite State Machine) [36, 16], and, therefore, each line code has a distinct color. The system is large and there is no way to generate a complete reachability graph. A probabilistic algorithm is applied to obtain a subgraph for verification and testing so that a covering set of paths will provide a complete test set for the system [16].

The INAP machine has 140 control states, 187 variables, and 541 lines of specification code to be covered by tests (hence 541 colors). The (probabilistically generated) reachability graph has $296,423$ nodes and $298,244$ edges. As can be easily observed, the graph is very sparse and a lot of nodes have degree 2. A straightforward application of our procedure generate 83 tests which have a complete coverage of all the specification code.

4.3 ATM PNNI

We applied **PIDTHIA** to test generation of an ATM network routing protocol, the ATM Forum Private Network-Network Interface (PNNI) Specification Version 1.0 [8]. It consists of three layers of protocols: the Hello protocol for identifying the status of PNNI's, the Database Synchronization protocol for maintenance of routing databases, and the Peer Group Leader Election protocol for operations of hierarchical routing. Since it is a hierarchical protocol, we can group the nodes into two peer groups, and each peer group then behaves like a single node. We use this two-node network connected by a full-duplex channel, instead of an isolated node/protocol, for test generation.

We report experimental results on Hello protocol. There are two versions: a simplified one and the original one [11]. For the simplified version, the reachability graph contains $6,079$ nodes and $7,584$ edges. It is a sparse graph of $1,850$ SCC's and most of them are singleton node SCC. Five tests cover all the specification code lines.

For the more complex original version, the reachability graph contains $60,230$ nodes and $68,140$ edges. It is a sparse graph of $8,122$ SCC's and most of them are singleton node SCC. Again five tests cover all the specification code lines.

For PHS the reachability graph is a DAG, and for INAP and PNNI it contains non-trivial SCC's. We "shrink" each SCC into a node with all the colors of the nodes and edges in the SCC. Algorithm 1-3 construct paths in the DAG's where some nodes are SCC's. To obtain valid test sequences we have to derive paths in the corresponding reachability graph.

Specifically, suppose that we have a path $\ldots, v_{i-1}, v_i, v_{i+1}, \ldots$ where v_i is an SCC of a graph G_i. When we "shrink" SCC G_i into a node v_i, examine the edge (v_{i-1}, u_i) which "enters" the SCC G_i and record the node u_i. Similarly, we record node u_{i+1} for the edge (u_{i+1}, v_{i+1}), which "exits" from the SCC. Suppose that the SCC inherits a color set C_i. We want to find a path in G_i from u_i to u_{i+1} such that all the colors in C_i are covered at least once. In general it is an NP-hard problem [12]. We use the following heuristic method. From u_i we find a node/edge with an uncovered color, using a breadth-first search for instance, and arrive at a node w_i. We delete the newly covered color and repeat the same process from w_i. After all the colors in C_i are included in the path, we take a shortest path to the exiting node u_{i+1}, and the total path is what is needed. The length of the path may not be optimal; the loss factor is at most $|G_i|$, i.e., the constructed path is of lenght at most $l^*|G_i|$ where l^* is the length of the optimal path and $|G_i|$ is the number of nodes in G_i. As indicated earlier, the test sequence length counts very little but the number of tests is crucial, and this approach is acceptable in practice.

5 Conclusion

We have presented the core of the automatic test generation tool **PYTHIA**: the test generation algorithms and their experiments on real system. They can be formulated as optimization problems on graphs. Most of these problems are NP-hard, and we proposed various heuristic methods that

perform well in practice.

Often system engineers come up with a number of tests based on their experiences or imagination that provide a good fault coverage. Their concern is: there is too much redundancy and, as a result, there are too many tests. They want to delete some tests without sacrificing the fault coverage. In this case, we have *test selection* rather than construction problems. Specifically, we have a complete test set, supposedly with a lot of redundancy. We want to select a subset of tests so that they are complete and has a minimal number: From a complete test set, select a complete subset of minimum cardinality. Obviously, it is the minimum Set Cover problem [18, 12] and is NP-hard. As mentioned in Section 3.2, we can use a greedy method by repeatedly choosing the test with a maximal number of uncovered colors until all the colors are covered. Similar to path construction problems, it is known that it is optimal within a *log* factor of the exact number of minimal complete tests and that one cannot do better in polynomial time in general.

While testing of FSM's and the control portions of protocols is a well studied problem, testing of EFSM's and the data portions of protocols is still at an early stage; the difficulties are from the state explosion due to the large number of combinations of variable values. For systems of the size of PHS, the problem is still manageable. However, for systems such as INAP and PNNI it is impossible to generate a reachability graph. The following minimization procedure is intended to reduce the size of the reachability graph for test generation and system analysis. Given a reachability graph from an EFSM, some nodes (configurations) may be equivalent and we want to minimize it by collapsing equivalent nodes first before test generation. Furthermore, we want to construct a minimized system even without first constructing a reachability graph, which is often impractical. It is known that this on-line minimization can be achieved in time polynomial in the size of the minimized systems [23].

Communication systems usually have timers and testing of the temporal properties is necessary. However, timers have an infinite range of values and their behaviors are difficult to model by variables as in EFSM's. Still it is possible to reduce the problem to a finite domain [4] and polynomial time algorithms are obtained for the minimization and reachability analysis [23].

Communication systems often exhibit nondeterministic behaviors due to the invisible internal transitions and the presence of timers [14]. Testing of nondeterministic systems even only for FSM's seems to be hard [5].

Protocol entities often send/receive parameters to/from each other and the executions (predicates and actions of the transitions) may depend on the parameter values received. To properly model and analyze the system behaviors, we further extend the model of EFSM and study parameterized extended finite state machines instead.

Mihalis, please tell a story about PYTHIA here. As network systems become so complex and unreliable, system builders and service providers do hope that there were a real Pythia who could advise about the quality of service of the systems built. The software tool **PYTHIA** was built to generate test sequences that detect system faults - a way of *guessing* Pythia's advice.

References

[1] A. V. Aho, A. T. Dahbura, D. Lee, and M. U. Uyar (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.

[2] A.V. Aho, J. E. Hopcroft, and J. D. Ullman (1974) *The Design and Analysis of Computer Algorithms*, Addison-Wesley.

[3] N. Alon, R. Yuster, and U. Zwick (1995) Color-coding, *J. ACM*, Vol. 42, No. 4, pp. 844-856.

[4] R. Alur, C. Courcoubetis, and D. Dill (1990) Model Checking for Real-time Systems, *Proc. 5th IEEE Symp. on Logic in Computer Science*, pp. 414-425.

[5] R. Alur, C. Courcoubetis, and M. Yannakakis (1995) Distinguishing tests for nondeterministic and probabilistic machines", *Proc. 27th Ann. ACM Symp. on Theory of Computing*, pp. 363-372.

[6] S. Arora, C. Lund, R. Motwani, M. Sudan, and M. Szegedy (1992) Proof Verification and Intractability of Approximation Problems, *Proc. of the 3rd IEEE Symp. on Foundations of Computer Science,* pp. 14-23.

[7] ANSI (1989) International standard ISO 8802-2, ANSI/IEEE std 802.2.

[8] PNNI 1.0 (1996) The ATM Forum Technical Committee Private Network-Network Interface Specification Version 1.0, af-pnni-0055.000.

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

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

[11] D. Cypher, D. Lee, M. Martin-Villaba, C. Prins, and D. Su (1998) Formal Specification, Verification, and Automatic Test Generation of ATM Routing Protocol: PNNI, PSTV-FORTE Tool Demonstration Proc.

[12] M. R. Garey and D. S. Johnson (1979) *Computers and Intractability: a Guide to the Theory of NP-Completeness,* W. H. Freeman.

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

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

[15] J. E. Hopcroft and J. D. Ullman (1979) *Introduction to Automata Theory, Languages, and Computation,* Addison-Wesley.

[16] S. Huang, D. Lee, and M. Staskauskas (1996) Validation-based Test Sequence Generation for Networks of Extended Finite State Machines, *Proc. FORTE/PSTV*, North Holland, R. Gotzhein Ed.

[17] D. S. Johnson (1974) Approximation algorithms for Combinatorial Problems, *J. of Computer and System Sciences,* Vol. 9, pp. 256-278.

[18] R. M. Karp (1972) Reducibility among Combinatorial Problems, *Complexity of Computer Computations*, R. E. Miller and J. W. Thatcher ED., Plenum Press, pp. 85-103.

[19] Z. Kohavi (1978) *Switching and Finite Automata Theory,* 2nd Ed., McGraw-Hill.

[20] E. L. Lawler (1976) *Combinatorial Optimization: Networks and Matroids*, Holt, Einehart and Winston, New York.

[21] D. Lee, D. Su, L. Collica and N. Golmie (1997) Conformance Test Suite for the ABR Rate Control Scheme in TM v4.0, *ATM Forum/97-0034*, February.

[22] D. Lee and M. Yannakakis (1994) Testing finite state machines: state identification and verification, *IEEE Trans. on Computers*, Vol. 43, No. 3, pp. 306-320.

[23] D. Lee and M. Yannakakis (1996a) Principles and Methods of Testing Finite State Machines - a Survey, *The Proceedings of IEEE*, Vol. 84, No. 8, pp. 1089-1123, August.

[24] D. Lee and M. Yannakakis (1996b) Optimization Problems from Feature Testing of Communication Protocols, *The Proc. of ICNP*, pp. 66-75.

[25] L. Lovasz (1975) On the Ratio of Optimal Integral and Fractional Covers," *Discrete Mathematics,* Vol. 13, pp. 383-390.

[26] L. Lovasz (1979) *Combinatorial Problems and Exercises,* North Holland.

[27] C. Lund and M. Yannakakis (1994) On the Hardness of Approximation Minimization Problems, *J. ACM*, Vol. 41, No. 5, pp. 960-981.

[28] R. E. Miller and S. Paul (1993) On the generation of minimal length test sequences for conformance testing of communication protocols, *IEEE/ACM Trans. on Networking,* Vol. 1, No. 1, pp. 116-129.

[29] E. F. Moore (1956) Gedanken-experiments on sequential machines, *Automata Studies,* Annals of Mathematics Studies, Princeton University Press, no. 34, pp. 129-153.

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

[31] C. H. Papadimitriou and M. Yannakakis (1991) Optimization, Approximation and Complexity Classes, *J. Comp. Syst. Sciences*, Vol. 43, No. 3, pp. 425-440.

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

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

[34] J. P. Schmidt and A. Siegel (1990) The Spatial Complexity of Oblivious k-probe Hash Function, *SIAM J. Comput.*, Vol. 19, No. 5, pp. 775-786.

[35] M. P. Vasilevskii (1973) Failure diagnosis of automata, *Kibernetika*, no. 4, pp. 98-108.

[36] F. Wagner (1992) VFSM Executable Specification, *Proc. CompEuro.*

[37] http://www-zoo.research.bell-labs.com/ nr/11341/pythia/

[38] B. Yang and H. Ural (1990) Protocol conformance test generation using multiple UIO sequences with overlapping, *Proc. SIGCOM*, pp. 118-125.

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

[40] D. M. Yellin and C. S. Jutla (1993) Finding Extremal Sets in Less than Quadratic Time, *Inf. Proc. Let.*, Vol. 48, pp. 29-34.